**Windows PowerShell**

**Language Specification**

**Version 2.0**

Notice

© 2009-2011 Microsoft Corporation. All rights reserved.

Microsoft, Windows, and Windows PowerShell are either registered trademarks or trademarks of Microsoft Corporation in the U.S.A. and/or other countries/regions.

Other product and company names mentioned herein may be the trademarks of their respective owners.

Contents

[1. Introduction 1](#_Toc263956120)

[2. Lexical Structure 2](#_Toc263956121)

[2.1 Grammars 2](#_Toc263956122)

[2.2 Lexical analysis 2](#_Toc263956123)

[2.2.1 Scripts 2](#_Toc263956124)

[2.2.2 Line terminators 3](#_Toc263956125)

[2.2.3 Comments 3](#_Toc263956126)

[2.2.4 White space 5](#_Toc263956127)

[2.3 Tokens 5](#_Toc263956128)

[2.3.1 Keywords 6](#_Toc263956129)

[2.3.2 Variables 6](#_Toc263956130)

[2.3.2.1 User-defined variables 8](#_Toc263956131)

[2.3.2.2 Automatic variables 8](#_Toc263956132)

[2.3.2.3 Preference variables 14](#_Toc263956133)

[2.3.3 Commands 17](#_Toc263956134)

[2.3.4 Parameters 17](#_Toc263956135)

[2.3.5 Literals 19](#_Toc263956136)

[2.3.5.1 Numeric literals 19](#_Toc263956137)

[2.3.5.1.1 Integer literals 20](#_Toc263956138)

[2.3.5.1.2 Real literals 21](#_Toc263956139)

[2.3.5.1.3 Multiplier suffixes 22](#_Toc263956140)

[2.3.5.2 String literals 22](#_Toc263956141)

[2.3.5.3 Null literal 28](#_Toc263956142)

[2.3.5.4 Boolean literals 28](#_Toc263956143)

[2.3.5.5 Array literals 28](#_Toc263956144)

[2.3.5.6 Hash literals 28](#_Toc263956145)

[2.3.5.7 Type names 28](#_Toc263956146)

[2.3.6 Operators and punctuators 29](#_Toc263956147)

[2.3.7 Escaped characters 30](#_Toc263956148)

[3. Basic concepts 31](#_Toc263956149)

[3.1 Providers and drives 31](#_Toc263956150)

[3.1.1 Aliases 32](#_Toc263956151)

[3.1.2 Environment variables 32](#_Toc263956152)

[3.1.3 File system 32](#_Toc263956153)

[3.1.4 Functions 33](#_Toc263956154)

[3.1.5 Variables 33](#_Toc263956155)

[3.2 Working locations 33](#_Toc263956156)

[3.3 Items 34](#_Toc263956157)

[3.4 Path names 34](#_Toc263956158)

[3.5 Scopes 36](#_Toc263956159)

[3.5.1 Introduction 36](#_Toc263956160)

[3.5.2 Scope names and numbers 37](#_Toc263956161)

[3.5.3 Variable name scope 38](#_Toc263956162)

[3.5.4 Function name scope 39](#_Toc263956163)

[3.5.5 Dot source notation 39](#_Toc263956164)

[3.5.6 Modules 39](#_Toc263956165)

[3.6 ReadOnly and Constant Properties 39](#_Toc263956166)

[3.7 Method overloads and call resolution 39](#_Toc263956167)

[3.7.1 Introduction 39](#_Toc263956168)

[3.7.2 Method overload resolution 40](#_Toc263956169)

[3.7.3 Applicable method 41](#_Toc263956170)

[3.7.4 Better method 41](#_Toc263956171)

[3.7.5 Better conversion 42](#_Toc263956172)

[3.8 Name lookup 44](#_Toc263956173)

[3.9 Type name lookup 44](#_Toc263956174)

[3.10 Automatic memory management 44](#_Toc263956175)

[3.11 Execution order 44](#_Toc263956176)

[3.12 Error handling 45](#_Toc263956177)

[3.13 Pipelines 45](#_Toc263956178)

[3.14 Modules 46](#_Toc263956179)

[3.15 Wildcard expressions 46](#_Toc263956180)

[3.16 Regular expressions 46](#_Toc263956181)

[4. Types 49](#_Toc263956182)

[4.1 Special types 50](#_Toc263956183)

[4.1.1 The void type 50](#_Toc263956184)

[4.1.2 The null type 50](#_Toc263956185)

[4.1.3 The object type 50](#_Toc263956186)

[4.2 Value types 50](#_Toc263956187)

[4.2.1 Boolean 50](#_Toc263956188)

[4.2.2 Character 50](#_Toc263956189)

[4.2.3 Integer 51](#_Toc263956190)

[4.2.4 Real number 52](#_Toc263956191)

[4.2.4.1 float and double 52](#_Toc263956192)

[4.2.4.2 decimal 53](#_Toc263956193)

[4.2.5 The switch type 54](#_Toc263956194)

[4.2.6 Enumeration types 54](#_Toc263956195)

[4.2.6.1 Action-Preference type 54](#_Toc263956196)

[4.2.6.2 Confirm-Impact type 54](#_Toc263956197)

[4.2.6.3 File-Attributes type 55](#_Toc263956198)

[4.2.6.4 Regular-Expression-Option type 56](#_Toc263956199)

[4.3 Reference types 56](#_Toc263956200)

[4.3.1 Strings 56](#_Toc263956201)

[4.3.2 Arrays 57](#_Toc263956202)

[4.3.3 Hashtables 58](#_Toc263956203)

[4.3.4 The xml type 58](#_Toc263956204)

[4.3.5 The regex type 58](#_Toc263956205)

[4.3.6 The ref type 58](#_Toc263956206)

[4.3.7 The scriptblock type 59](#_Toc263956207)

[4.3.8 The math type 60](#_Toc263956208)

[4.4 Generic types 62](#_Toc263956209)

[4.5 Anonymous types 62](#_Toc263956210)

[4.5.1 Provider description type 62](#_Toc263956211)

[4.5.2 Drive description type 63](#_Toc263956212)

[4.5.3 Variable description type 63](#_Toc263956213)

[4.5.4 Alias description type 64](#_Toc263956214)

[4.5.5 Working location description type 66](#_Toc263956215)

[4.5.6 Environment variable description type 66](#_Toc263956216)

[4.5.7 Application description type 67](#_Toc263956217)

[4.5.8 Cmdlet description type 68](#_Toc263956218)

[4.5.9 External script description type 69](#_Toc263956219)

[4.5.10 Function description type 70](#_Toc263956220)

[4.5.11 Filter description type 72](#_Toc263956221)

[4.5.12 Module description type 73](#_Toc263956222)

[4.5.13 Custom object description type 73](#_Toc263956223)

[4.5.14 Command description type 73](#_Toc263956224)

[4.5.15 Error record description type 74](#_Toc263956225)

[4.5.16 Enumerator description type 74](#_Toc263956226)

[4.5.17 Directory description type 75](#_Toc263956227)

[4.5.18 File description type 76](#_Toc263956228)

[4.5.19 Date-Time description type 77](#_Toc263956229)

[4.5.20 Group-Info description type 78](#_Toc263956230)

[4.5.21 Generic-Measure-Info description type 78](#_Toc263956231)

[4.5.22 Text-Measure-Info description type 79](#_Toc263956232)

[4.5.23 Credential type 80](#_Toc263956233)

[4.5.24 Method designator type 80](#_Toc263956234)

[4.5.25 Member definition type 80](#_Toc263956235)

[4.6 Type extension and adaptation 81](#_Toc263956236)

[5. Variables 83](#_Toc263956237)

[5.1 Writable location 83](#_Toc263956238)

[5.2 Variable categories 83](#_Toc263956239)

[5.2.1 Static variables 84](#_Toc263956240)

[5.2.2 Instance variables 84](#_Toc263956241)

[5.2.3 Array elements 84](#_Toc263956242)

[5.2.4 Hashtable key/value pairs 85](#_Toc263956243)

[5.2.5 Parameters 85](#_Toc263956244)

[5.2.6 Ordinary variables 85](#_Toc263956245)

[5.2.7 Variables on provider drives 85](#_Toc263956246)

[5.3 Constrained variables 85](#_Toc263956247)

[6. Conversions 87](#_Toc263956248)

[6.1 Conversion to void 87](#_Toc263956249)

[6.2 Conversion to bool 87](#_Toc263956250)

[6.3 Conversion to char 87](#_Toc263956251)

[6.4 Conversion to integer 88](#_Toc263956252)

[6.5 Conversion to float and double 88](#_Toc263956253)

[6.6 Conversion to decimal 89](#_Toc263956254)

[6.7 Conversion to object 89](#_Toc263956255)

[6.8 Conversion to string 89](#_Toc263956256)

[6.9 Conversion to array 90](#_Toc263956257)

[6.10 Conversion to xml 90](#_Toc263956258)

[6.11 Conversion to regex 90](#_Toc263956259)

[6.12 Conversion to scriptblock 91](#_Toc263956260)

[6.13 Conversion to enumeration types 91](#_Toc263956261)

[6.14 Conversion to other reference types 91](#_Toc263956262)

[6.15 Usual arithmetic conversions 91](#_Toc263956263)

[6.16 Conversion from string to numeric type 92](#_Toc263956264)

[6.17 Conversion during parameter binding 92](#_Toc263956265)

[6.18 .NET Conversion 93](#_Toc263956266)

[7. Expressions 95](#_Toc263956267)

[7.1 Primary expressions 96](#_Toc263956268)

[7.1.1 Grouping parentheses 96](#_Toc263956269)

[7.1.2 Member access 98](#_Toc263956270)

[7.1.3 Invocation expressions 98](#_Toc263956271)

[7.1.4 Element access 99](#_Toc263956272)

[7.1.4.1 Subscripting an array 99](#_Toc263956273)

[7.1.4.2 Subscripting a string 100](#_Toc263956274)

[7.1.4.3 Subscripting a Hashtable 101](#_Toc263956275)

[7.1.4.4 Subscripting an XML document 101](#_Toc263956276)

[7.1.4.5 Generating array slices 102](#_Toc263956277)

[7.1.5 Postfix increment and decrement operators 102](#_Toc263956278)

[7.1.6 $(…) operator 103](#_Toc263956279)

[7.1.7 @(…) operator 104](#_Toc263956280)

[7.1.8 Script block expression 104](#_Toc263956281)

[7.1.9 Hash literal expression 105](#_Toc263956282)

[7.1.10 Type literal expression 106](#_Toc263956283)

[7.2 Unary operators 107](#_Toc263956284)

[7.2.1 Unary comma operator 107](#_Toc263956285)

[7.2.2 Logical NOT 108](#_Toc263956286)

[7.2.3 Bitwise NOT 108](#_Toc263956287)

[7.2.4 Unary plus 108](#_Toc263956288)

[7.2.5 Unary minus 109](#_Toc263956289)

[7.2.6 Prefix increment and decrement operators 109](#_Toc263956290)

[7.2.7 The unary -join operator 110](#_Toc263956291)

[7.2.8 The unary -split operator 110](#_Toc263956292)

[7.2.9 Cast operator 110](#_Toc263956293)

[7.3 Binary comma operator 111](#_Toc263956294)

[7.4 Range operator 111](#_Toc263956295)

[7.5 Format operator 112](#_Toc263956296)

[7.6 Multiplicative operators 113](#_Toc263956297)

[7.6.1 Multiplication 113](#_Toc263956298)

[7.6.2 String replication 113](#_Toc263956299)

[7.6.3 Array replication 113](#_Toc263956300)

[7.6.4 Division 114](#_Toc263956301)

[7.6.5 Remainder 114](#_Toc263956302)

[7.7 Additive operators 115](#_Toc263956303)

[7.7.1 Addition 115](#_Toc263956304)

[7.7.2 String concatentaion 115](#_Toc263956305)

[7.7.3 Array concatenation 115](#_Toc263956306)

[7.7.4 Hashtable concatenation 116](#_Toc263956307)

[7.7.5 Subtraction 116](#_Toc263956308)

[7.8 Comparison operators 117](#_Toc263956309)

[7.8.1 Equality and relational operators 117](#_Toc263956310)

[7.8.2 Containment operators 118](#_Toc263956311)

[7.8.3 Type testing and conversion operators 118](#_Toc263956312)

[7.8.4 Pattern matching and text manipulation operators 119](#_Toc263956313)

[7.8.4.1 The -like and -notlike operators 119](#_Toc263956314)

[7.8.4.2 The -match and -notmatch operators 119](#_Toc263956315)

[7.8.4.3 The -replace operator 120](#_Toc263956316)

[7.8.4.4 The binary -join operator 120](#_Toc263956317)

[7.8.4.5 The binary -split operator 121](#_Toc263956318)

[7.8.4.6 Submatches 122](#_Toc263956319)

[7.9 Bitwise operators 123](#_Toc263956320)

[7.10 Logical operators 123](#_Toc263956321)

[7.11 Assignment operators 124](#_Toc263956322)

[7.11.1 Simple assignment 125](#_Toc263956323)

[7.11.2 Compound assignment 126](#_Toc263956324)

[7.12 Redirection operators 126](#_Toc263956325)

[8. Statements 128](#_Toc263956326)

[8.1 Statement blocks and lists 128](#_Toc263956327)

[8.1.1 Labeled statements 128](#_Toc263956328)

[8.1.2 Statement values 129](#_Toc263956329)

[8.2 Pipeline statements 130](#_Toc263956330)

[8.3 The if statement 132](#_Toc263956331)

[8.4 Iteration statements 133](#_Toc263956332)

[8.4.1 The while statement 133](#_Toc263956333)

[8.4.2 The do statement 133](#_Toc263956334)

[8.4.3 The for statement 134](#_Toc263956335)

[8.4.4 The foreach statement 135](#_Toc263956336)

[8.5 Flow control statements 136](#_Toc263956337)

[8.5.1 The break statement 136](#_Toc263956338)

[8.5.2 The continue statement 137](#_Toc263956339)

[8.5.3 The throw statement 138](#_Toc263956340)

[8.5.4 The return statement 139](#_Toc263956341)

[8.5.5 The exit statement 140](#_Toc263956342)

[8.6 The switch statement 140](#_Toc263956343)

[8.7 The try/finally statement 143](#_Toc263956344)

[8.8 The trap statement 145](#_Toc263956345)

[8.9 The data statement 146](#_Toc263956346)

[8.10 Function definitions 148](#_Toc263956347)

[8.10.1 Filter functions 149](#_Toc263956348)

[8.10.2 Argument processing 149](#_Toc263956349)

[8.10.3 Parameter initializers 150](#_Toc263956350)

[8.10.4 The [switch] type constraint 150](#_Toc263956351)

[8.10.5 Pipelines and functions 150](#_Toc263956352)

[8.10.6 Named blocks 151](#_Toc263956353)

[8.10.7 dynamicParam block 151](#_Toc263956354)

[8.10.8 param block 152](#_Toc263956355)

[8.11 Parameter binding 153](#_Toc263956356)

[9. Arrays 155](#_Toc263956357)

[9.1 Introduction 155](#_Toc263956358)

[9.2 Array creation 155](#_Toc263956359)

[9.3 Array concatenation 156](#_Toc263956360)

[9.4 Constraining element types 156](#_Toc263956361)

[9.5 Arrays as reference types 157](#_Toc263956362)

[9.6 Arrays as array elements 157](#_Toc263956363)

[9.7 Negative subscripting 158](#_Toc263956364)

[9.8 Bounds checking 158](#_Toc263956365)

[9.9 Array slices 158](#_Toc263956366)

[9.10 Copying an array 158](#_Toc263956367)

[9.11 Enumerating over an array 158](#_Toc263956368)

[9.12 Multidimensional array flattening 159](#_Toc263956369)

[10. Hashtables 160](#_Toc263956370)

[10.1 Introduction 160](#_Toc263956371)

[10.2 Hashtable creation 161](#_Toc263956372)

[10.3 Adding and removing Hashtable elements 161](#_Toc263956373)

[10.4 Hashtable concatenation 161](#_Toc263956374)

[10.5 Hashtables as reference types 161](#_Toc263956375)

[10.6 Enumerating over a Hashtable 161](#_Toc263956376)

[11. Modules 162](#_Toc263956377)

[11.1 Introduction 162](#_Toc263956378)

[11.2 Writing a script module 162](#_Toc263956379)

[11.3 Installing a script module 163](#_Toc263956380)

[11.4 Importing a script module 163](#_Toc263956381)

[11.5 Removing a script module 163](#_Toc263956382)

[11.6 Module manifests 164](#_Toc263956383)

[11.7 Dynamic modules 168](#_Toc263956384)

[11.8 Closures 169](#_Toc263956385)

[12. Attributes 170](#_Toc263956386)

[12.1 Attribute specification 170](#_Toc263956387)

[12.2 Attribute instances 170](#_Toc263956388)

[12.3 Reserved attributes 170](#_Toc263956389)

[12.3.1 The Alias attribute 171](#_Toc263956390)

[12.3.2 The AllowEmptyCollection attribute 171](#_Toc263956391)

[12.3.3 The AllowEmptyString attribute 171](#_Toc263956392)

[12.3.4 The AllowNull attribute 172](#_Toc263956393)

[12.3.5 The CmdletBinding attribute 172](#_Toc263956394)

[12.3.6 The OutputType attribute 173](#_Toc263956395)

[12.3.7 The Parameter attribute 174](#_Toc263956396)

[12.3.8 The ValidateCount attribute 178](#_Toc263956397)

[12.3.9 The ValidateLength attribute 179](#_Toc263956398)

[12.3.10 The ValidateNotNull attribute 179](#_Toc263956399)

[12.3.11 The ValidateNotNullOrEmpty attribute 179](#_Toc263956400)

[12.3.12 The ValidatePattern attribute 180](#_Toc263956401)

[12.3.13 The ValidateRange attribute 180](#_Toc263956402)

[12.3.14 The ValidateScript attribute 181](#_Toc263956403)

[12.3.15 The ValidateSet attribute 182](#_Toc263956404)

[13. Cmdlets 183](#_Toc263956405)

[13.1 Add-Content (alias ac) 183](#_Toc263956406)

[13.2 Add-Member 185](#_Toc263956407)

[13.3 Clear-Content (alias clc) 187](#_Toc263956408)

[13.4 Clear-Item (alias cli) 189](#_Toc263956409)

[13.5 Clear-Variable (alias clv) 191](#_Toc263956410)

[13.6 Compare-Object (alias compare) 193](#_Toc263956411)

[13.7 ConvertFrom-StringData 195](#_Toc263956412)

[13.8 Convert-Path (alias cvpa) 196](#_Toc263956413)

[13.9 Copy-Item (alias copy, cp, cpi) 197](#_Toc263956414)

[13.10 Export-Alias (alias epal) 199](#_Toc263956415)

[13.11 Export-ModuleMember 201](#_Toc263956416)

[13.12 ForEach-Object (alias %, foreach) 203](#_Toc263956417)

[13.13 Get-Alias (alias gal) 204](#_Toc263956418)

[13.14 Get-ChildItem (alias dir, gci, ls) 205](#_Toc263956419)

[13.15 Get-Command (alias gcm) 207](#_Toc263956420)

[13.16 Get-Content (alias cat, gc, type) 210](#_Toc263956421)

[13.17 Get-Credential 211](#_Toc263956422)

[13.18 Get-Date 212](#_Toc263956423)

[13.19 Get-Help (alias help, man) 216](#_Toc263956424)

[13.20 Get-Item (alias gi) 218](#_Toc263956425)

[13.21 Get-Location (alias gl, pwd) 220](#_Toc263956426)

[13.22 Get-Member (alias gm) 221](#_Toc263956427)

[13.23 Get-Module (alias gmo) 224](#_Toc263956428)

[13.24 Get-PSDrive (alias gdr) 225](#_Toc263956429)

[13.25 Get-PSProvider 226](#_Toc263956430)

[13.26 Get-Variable (alias gv) 227](#_Toc263956431)

[13.27 Group-Object (alias group) 228](#_Toc263956432)

[13.28 Import-Module (alias ipmo) 230](#_Toc263956433)

[13.29 Invoke-Item (alias ii) 234](#_Toc263956434)

[13.30 Join-Path 235](#_Toc263956435)

[13.31 Measure-Object (alias measure) 237](#_Toc263956436)

[13.32 Move-Item (alias mi, move, mv) 239](#_Toc263956437)

[13.33 New-Alias (alias nal) 241](#_Toc263956438)

[13.34 New-Item (alias ni) 243](#_Toc263956439)

[13.35 New-Module (alias nmo) 245](#_Toc263956440)

[13.36 New-Object 247](#_Toc263956441)

[13.37 New-Variable (alias nv) 249](#_Toc263956442)

[13.38 Pop-Location (alias popd) 251](#_Toc263956443)

[13.39 Push-Location (alias pushd) 252](#_Toc263956444)

[13.40 Remove-Item (alias del, erase, rd, ri, rm, rmdir) 253](#_Toc263956445)

[13.41 Remove-Module (alias rmo) 255](#_Toc263956446)

[13.42 Remove-Variable (alias rv) 257](#_Toc263956447)

[13.43 Rename-Item (alias ren, rni) 258](#_Toc263956448)

[13.44 Resolve-Path (alias rvpa) 260](#_Toc263956449)

[13.45 Select-Object (alias select) 261](#_Toc263956450)

[13.46 Set-Alias (alias sal) 263](#_Toc263956451)

[13.47 Set-Content (alias sc) 265](#_Toc263956452)

[13.48 Set-Item (alias si) 267](#_Toc263956453)

[13.49 Set-Location (alias cd, chdir, sl) 270](#_Toc263956454)

[13.50 Set-Variable (alias set, sv) 271](#_Toc263956455)

[13.51 Sort-Object (alias sort) 273](#_Toc263956456)

[13.52 Split-Path 275](#_Toc263956457)

[13.53 Tee-Object (alias tee) 277](#_Toc263956458)

[13.54 Test-Path 278](#_Toc263956459)

[13.55 Where-Object (alias ?, where) 280](#_Toc263956460)

[13.56 Common parameters 281](#_Toc263956461)

[A. Comment-Based Help 284](#_Toc263956462)

[A.1 Introduction 284](#_Toc263956463)

[A.2 Help directives 285](#_Toc263956464)

[A.2.1 .DESCRIPTION 285](#_Toc263956465)

[A.2.2 .EXAMPLE 285](#_Toc263956466)

[A.2.3 .EXTERNALHELP 285](#_Toc263956467)

[A.2.4 .FORWARDHELPCATEGORY 286](#_Toc263956468)

[A.2.5 .FORWARDHELPTARGETNAME 286](#_Toc263956469)

[A.2.6 .INPUTS 286](#_Toc263956470)

[A.2.7 .LINK 287](#_Toc263956471)

[A.2.8 .NOTES 287](#_Toc263956472)

[A.2.9 *.*OUTPUTS 287](#_Toc263956473)

[A.2.10 .PARAMETER 288](#_Toc263956474)

[A.2.11 .SYNOPSIS 289](#_Toc263956475)

[B. Grammar 290](#_Toc263956476)

[B.1 Lexical grammar 290](#_Toc263956477)

[B.1.1 Line terminators 290](#_Toc263956478)

[B.1.2 Comments 290](#_Toc263956479)

[B.1.3 White space 291](#_Toc263956480)

[B.1.4 Tokens 291](#_Toc263956481)

[B.1.5 Keywords 292](#_Toc263956482)

[B.1.6 Variables 292](#_Toc263956483)

[B.1.7 Commands 293](#_Toc263956484)

[B.1.8 Literals 293](#_Toc263956485)

[Integer Literals 294](#_Toc263956486)

[Real Literals 294](#_Toc263956487)

[String Literals 294](#_Toc263956488)

[B.1.9 Simple Names 296](#_Toc263956489)

[B.1.10 Type Names 297](#_Toc263956490)

[B.1.11 Operators and punctuators 297](#_Toc263956491)

[B.2 Syntactic grammar 298](#_Toc263956492)

[B.2.1 Basic concepts 298](#_Toc263956493)

[B.2.2 Statements 298](#_Toc263956494)

[B.2.3 Expressions 303](#_Toc263956495)

[B.2.4 Attributes 307](#_Toc263956496)

[C. References 308](#_Toc263956497)

# Introduction

PowerShell is a command-line shell and scripting language, designed especially for system administrators.

Most shells operate by executing a command or utility in a new process, and presenting the results to the user as text. These shells also have commands that are built into the shell and run in the shell process. Because there are few built-in commands, many utilities have been created to supplement them. PowerShell is very different. Instead of processing text, the shell processes objects. PowerShell also includes a large set of built-in commands with each having a consistent interface and these can work with user-written commands.

An object is a data entity that has properties (i.e., characteristics) and methods (i.e., actions that can be performed on the object). All objects of the same type have the same base set of properties and methods, but each instance of an object can have different property values.

A major advantage of using objects is that it is much easier to pipeline commands; that is, to write the output of one command to another command as input. (In a traditional command-line environment, the text output from one command needs to be manipulated to meet the input format of another.)

PowerShell includes a very rich scripting language that supports constructs for looping, conditions, flow-control, and variable assignment. This language has syntax features and keywords similar to those used in the C# programming language (§C).

There are four kinds of commands in PowerShell: scripts, functions and methods, cmdlets, and native commands.

* A file of commands is called a script. [Note: By convention, a script has a filename extension of .ps1. end note] The top-most level of a PowerShell program is a script, which, in turn, can invoke other commands.
* PowerShell supports modular programming via named procedures. A procedure written in PowerShell is called a function, while an external procedure made available by the execution environment (and typically written in some other language) is called a method.
* A cmdlet—pronounced "command-let"—is a simple, single-task command-line tool. Although a cmdlet can be used on its own, the full power of cmdlets is realized when they are used in combination to perform complex tasks.
* A native command is a command that is built in to the host environment.

Each time the PowerShell runtime environment begins execution, it begins what is called a session. Commands then execute within the context of that session.

This specification defines the PowerShell language, the built-in cmdlets, and the use of objects via the pipeline.

Windows PowerShell: Unlike most shells, which accept and return text, Windows PowerShell is built on top of the .NET Framework common language runtime (CLR) and the .NET Framework, and accepts and returns .NET Framework objects.

# Lexical Structure

## Grammars

This specification shows the syntax of the PowerShell language using two grammars. The lexical grammar (§B.1) shows how Unicode characters are combined to form line terminators, comments, white space, and tokens. The syntactic grammar (§B.2) shows how the tokens resulting from the lexical grammar are combined to form PowerShell scripts.

For convenience, fragments of these grammars are replicated in appropriate places throughout this specification.

Any use of the characters ‘a’ through ‘z’ in the grammars is case insensitive. [Note: This means that letter case in variables, aliases, function names, keywords, statements, and operators is ignored. However, throughout this specification, such names are written in lowercase, except for some automatic and preference variables. end note]

## Lexical analysis

### Scripts

Syntax:

input:  
input-elementsopt signature-blockopt

input-elements:  
input-element  
input-elements input-element

input-element:  
whitespace  
comment  
token

signature-block:  
signature-begin signature signature-end

signature-begin:  
new-line-character # SIG # Begin signature block new-line-character

signature:  
base64 encoded signature blob in multiple single-line-comments

signature-end:  
new-line-character # SIG # End signature block new-line-character

Description:

The input source stream to a PowerShell translator is the input in a script, which contains a sequence of Unicode characters. The lexical processing of this stream involves the reduction of those characters into a sequence of tokens, which go on to become the input of syntactic analysis.

A script is a group of PowerShell commands stored in a script-file. The script itself has no name, per se, and takes its name from its source file. The end of that file indicates the end of the script.

A script may optionally contain a digital signature. A host environment is not required to process any text that follows a signature or anything that looks like a signature. The creation and use of digital signatures are not covered by this specification.

### Line terminators

Syntax:

new-line-character:  
Carriage return character (U+000D)  
Line feed character (U+000A)  
Carriage return character (U+000D) followed by line feed character (U+000A)

new-lines:  
new-line-character  
new-lines new-line-character

Description:

The presence of new-line-characters in the input source stream divides it into lines that can be used for such things as error reporting and the detection of the end of a single-line comment.

A line terminator can be treated as white space (§2.2.4).

### Comments

Syntax:

comment:  
single-line-comment  
requires-comment  
delimited-comment

single-line-comment:  
# input-charactersopt

input-characters:  
input-character  
input-characters input-character

input-character:  
Any Unicode character except a new-line-character

requires-comment:  
#requires whitespace command-arguments

dash:  
- (U+002D)  
EnDash character (U+2013)  
EmDash character (U+2014)  
Horizontal bar character (U+2015)

dashdash:  
dash dash

delimited-comment:  
<# delimited-comment-textopt hashes >

delimited-comment-text:  
delimited-comment-section  
delimited-comment-text delimited-comment-section

delimited-comment-section:  
>  
hashesopt not-greater-than-or-hash

hashes:  
#  
hashes #

not-greater-than-or-hash:  
Any Unicode character except > or #

Description:

Source code can be annotated by the use of comments.

A single-line-comment begins with the character # and ends with a new-line-character.

A delimited-comment begins with the character pair <# and ends with the character pair #>. It can occur as part of a source line, as a whole source line, or it can span any number of source lines.

A comment is treated as white space.

The productions above imply that

* Comments do not nest.
* The character sequences <# and #> have no special meaning in a single-line comment.
* The character # has no special meaning in a delimited comment.

The lexical grammar implies that comments cannot occur inside tokens.

(See §A for information about creating script files that contain special-valued comments that are used to generate documentation from script files.)

A requires-comment specifies the criteria that have to be met for its containing script to be allowed to run. The primary criterion is the version of PowerShell being used to run the script. The minimum version requirement is specified as follows:

#requires -Version N[.n]

Where N is the (required) major version and n is the (optional) minor version.

A requires-comment can be present in any script file; however, it cannot be present inside a function or cmdlet. It must be the first item on a source line. A script can contain multiple requires-comments.

A character sequence is only recognized as a comment if that sequence begins with # or <#. For example, hello#there is considered a single token whereas hello #there is considered the token hello followed by a single-line comment. As well as following white space, the comment start sequence can also be preceded by any expression-terminating or statement-terminating character (such as ), }, ], ', ", or ;).

Windows PowerShell: A requires-comment cannot be present inside a snap-in.

Windows PowerShell: There are two other forms of a requires-comment:

#requires –PsSnapIn PsSnapIn [ -Version N [.n ] ]  
#requires –ShellId ShellId

### White space

Syntax:

whitespace:  
Any character with Unicode class Zs, Zl, or Zp  
Horizontal tab character (U+0009)  
Vertical tab character (U+000B)  
Form feed character (U+000C)  
` (The backtick character U+0060) followed by new-line-character

Description:

White space consists of any sequence of one or more whitespace characters.

Except for the fact that white space may act as a separator for tokens, it is ignored.

Unlike some popular languages, PowerShell does not consider line-terminator characters (§2.2.2) to be white space. However, a line terminator can be treated as white space by preceding it immediately by a backtick character, ` (U+0060). This is necessary when the contents of a line are complete syntactically, yet the following line contains tokens intended to be associated with the previous line. For example,

$number = 10 # assigns 10 to $number; nothing is written to the pipeline  
+ 20 # writes 20 to the pipeline  
- 50 # writes -50 to the pipeline  
$number # writes $number's value, 10, to the pipeline

$number = 10 ` # backtick indicates the source line is continued  
+ 20 ` # backtick indicates the source line is continued  
- 50 # assigns -20 to $number; nothing is written to the pipeline  
$number # writes $number's value, -20, to the pipeline

## Tokens

Syntax:

token:  
keyword  
variable  
command  
command-parameter  
command-argument-token  
integer-literal  
real-literal  
string-literal  
type-literal  
operator-or-punctuator

Description:

A token is the smallest lexical element within the PowerShell language.

Tokens can be separated by new-lines, comments, white space, or any combination thereof.

### Keywords

Syntax:

keyword: one of  
begin break catch class  
continue data define do  
dynamicparam else elseif end  
exit filter finally for  
foreach from function if  
in param process return  
switch throw trap try  
until using var while

Description:

A keyword is a sequence of characters that has a special meaning when used in a context-dependent place. Most often, this is as the first token in a statement; however, there are other locations, as indicated by the grammar. (A token that looks like a keyword, but is not being used in a keyword context, is a command-name or a command-argument.)

The keywords class, define, from, using, and var are reserved for future use.

### Variables

Syntax:

variable:  
$$  
$?  
$^  
$ variable-scopeopt variable-characters  
@ variable-scopeopt variable-characters  
*braced-variable*

*braced-variable:*${ variable-scopeopt braced-variable-characters }

variable-scope:  
global:  
local:  
private:  
script:  
variable-namespace

variable-namespace:  
variable-characters :

variable-characters:  
variable-character  
variable-characters variable-character

variable-character:  
A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nd  
\_ (The underscore character U+005F)  
?

braced-variable-characters:  
braced-variable-character  
braced-variable-characters braced-variable-character

braced-variable-character:  
Any Unicode character except  
 } (The closing curly brace character U+007D)  
 ` (The backtick character U+0060)  
escaped-character

escaped-character:  
` (The backtick character U+0060) followed by any Unicode character

Description:

Variables are discussed in detail in (§5). The variable $? is discussed in §2.3.2.2. Scopes are discussed in §3.5.

The variables $$ and $^ are reserved for use in an interactive environment, which is outside the scope of this specification.

There are two ways of writing a variable name: A braced variable name, which begins with $, followed by a curly bracket-delimited set of one or more almost-arbitrary characters; and an ordinary variable name, which also begins with $, followed by a set of one or more characters from a more restrictive set than a braced variable name allows. Every ordinary variable name can be expressed using a corresponding braced variable name.

$totalCost  
$Maximum\_Count\_26

$végösszeg # Hungarian  
$итог # Russian  
$総計 # Japanese (Kanji)

${Maximum\_Count\_26}  
${Name with`twhite space and `{punctuation`}}  
${E:\File.txt}

There is no limit on the length of a variable name, all characters in a variable name are significant, and letter case is not distinct.

There are several different kinds of variables: user-defined (§2.3.2.1), automatic (§2.3.2.2), and preference (§2.3.2.3). They can all coexist in the same scope (§3.5).

Consider the following function definition and calls:

function Get-Power ([long]$base, [int]$exponent) { … }  
  
Get-Power 5 3 # $base is 5, $exponent is 3  
Get-Power -exponent 3 -base 5 # " " "

Each argument is passed by position or name, one at a time. However, a set of arguments can be passed as a group with expansion into individual arguments being handled by the runtime environment. This automatic argument expansion is known as splatting. For example,

$values = 5,3 # put arguments into an array  
Get-Power @values

$hash = @{ exponent = 3; base = 5 } # put arguments into a Hashtable  
Get-Power @hash

This is achieved by using @ instead of $ as the first character of the variable being passed. This notation can only be used in an argument to a command.

Names are partitioned into various namespaces each of which is stored on a virtual drive (§3.1). For example, variables are stored on Variable:, environment variables are stored on Env:, functions are stored on Function:, and aliases are stored on Alias:. All of these names can be accessed as variables using the variable-namespace production within variable-scope. For example,

function F { "Hello from F" }  
$Function:F # invokes function F

Set-Alias A F  
$Alias:A # invokes function F via A

$Count = 10  
$Variable:Count # accesses variable Count  
$Env:Path # accesses environment variable Path

Any use of a variable name with an explicit Variable: namespace is equivalent to the use of that same variable name without that qualification. For example, $v and $Variable:v are interchangeable.

As well as being defined in the language, variables can also be defined by the cmdlet New-Variable (§13.37).

#### User-defined variables

Any variable name allowed by the grammar but not used by automatic or preference variables is available for user-defined variables.

User-defined variables are created and managed by user-defined script.

#### Automatic variables

Automatic variables store state information about the PowerShell environment. Their values can be read in user-written script but not written.

| **Variable** | **Meaning** |
| --- | --- |
| $? | Contains the status of the last operation. It contains $true if the last operation succeeded, and $false if it failed. For more information about error handling see §3.12. Also see $LastExitCode. |
| $\_ | Exception handling: Within a matching catch clause (§8.7) or trap statement (§8.8), this variable contains an error record (§3.12), which contains a description of the current exception.  Filter and process block of a function: Within a filter (§8.10.1, §8.10.5), this variable provides access to the current object being processed from the input collection coming from a pipeline. In a named-block scenario, it is accessible in both the process and end blocks.  Script: argument passed to a script, function, filter, or cmdlet for a parameter having the ValidateScript attribute (§12.3.14).  Binary -split operator: When the right operand of this operator (§7.8.4.5) designates a script block, inside that script block $\_ represents each character in the input string(s), in lexical order, one character at a time.  Script block: Within a script block used as an argument to a cmdlet, this variable provides access to the current object being processed from the input collection coming from a pipeline.  Switch statement: Within a switch-clause statement-block (§8.6), this variable has the type and value of the switch-condition that caused control to go to that statement-block. |
| $args | Defined inside each function (§8.10), filter (§8.10.1), script, and script block (§8.11) as an unconstrained 1-dimensional array containing all arguments not bound by name or position, in lexical order. |
| $ConsoleFileName | Windows PowerShell: Contains the path of the console file (.psc1) that was most recently used in the session. This variable is populated when Windows PowerShell is started with the PSConsoleFile parameter or when the cmdlet Export-Console is used to export snap-in names to a console file.  When the cmdlet Export-Console is used without parameters, it automatically updates the console file that was most recently used in the session. This variable the file that will be updated. |
| $Error | Contains a collection of error records that represent the most recent errors. For more information, see §3.12. |
| $Event | Windows PowerShell: Contains a PSEventArgs object that represents the event being processed. This variable is populated only within the Action block of an event registration command, such as Register-ObjectEvent. The value of this variable is the same object returned by the cmdlet Get-Event. As such, the properties of $Event (such as $Event.TimeGenerated), can be used in an Action script block. |
| $EventSubscriber | Windows PowerShell: Contains a PSEventSubscriber object that represents the event subscriber of the event that is being processed. This variable is populated only within the Action block of an event registration command. The value of this variable is the same object returned by the cmdlet Get-EventSubscriber. |
| $ExecutionContext | Windows PowerShell: Contains an EngineIntrinsics object that represents the execution context of the Windows PowerShell host. This variable can be used to find the execution objects that are available to cmdlets. |
| $false | Has type bool (§4.2.1) and the constant value False. |
| $foreach | An enumerator created for any foreach statement. This variable exists only while the loop is executing. The type of an object that represents an enumerator is described in §4.5.16. |
| $Home | Contains the full path of the user's home directory. |
| $Host | Windows PowerShell: Contains an object that represents the current host application for Windows PowerShell. This variable can represent the current host in commands or to display or change the properties of the host, such as $Host.version or $Host.CurrentCulture, or $host.ui.rawui.setbackgroundcolor("Red"). |
| $input | An enumerator for a collection delivered to a function in a pipeline. The type of an object that represents an enumerator is described in §4.5.16. $input is only defined inside a process block (§8.10.6). |
| $LastExitCode | Whereas $? indicates success or failure, $LastExitCode contains the exit code of the last native command or script run. |
| $matches | Regular expression matching: Defined when the left operator of ‑match/‑notmatch or their variants (§7.8.4.2) is not a collection and the result is $true. $matches is a Hashtable whose keys are indexes that correspond to parts of the pattern that matched. The values are the substrings that matched.  Switch statement (§8.6): Defined inside the body of a matching pattern block when that pattern is a regular expression. Its value is the string that matched. |
| $MyInvocation | Windows PowerShell: Contains an object with information about the current command, such as a script, function, or script block. The information in this object, such as the path and file name of the script ($MyInvocation.MyCommand.Path) or the name of a function ($MyInvocation.MyCommand.Name), can be used to identify the current command. This is particularly useful for finding the name of the script that is running. |
| $NestedPromptLevel | Windows PowerShell: Contains the current prompt level. A value of 0 indicates the original prompt level. The value is incremented when a nested prompt level is entered and decremented when such a level it exited.  When a nested prompt level is entered, Windows PowerShell pauses the current command, saves the execution context, and increments the value of $NestedPromptLevel. To create additional nested command prompts (up to 128 levels) complete the command. To return to the previous command prompt level, enter exit.  For example, when prompted to confirm an action (such as the command Del –Confirm \*), the user is given the choice to Suspend. Suspend enters a nested prompt, and exiting the nested prompt returns back to the confirm choice prompt. |
| $null | The only instance of the null type (§4.1.2), and its value is constant. It is referred to as the null value. |
| $PID | Windows PowerShell: Contains the process identifier (PID) of the process that is hosting the current Windows PowerShell session. |
| $PsBoundParameters | Windows PowerShell: Contains a dictionary of the active parameters and their current values. This variable has a value only in a scope where parameters are declared, such as a script or function. It can be used to display or change the current values of parameters or to pass parameter values to another script or function. For example:  function test {  param ($a, $b)  # Display the parameters in dictionary format.  $PsBoundParameters  # Call the Test1 function with $a and $b.  Test1 @PsBoundParameters  } |
| $PsCmdlet | An object that represents the cmdlet or function being executed. See §4.5.14. |
| $PsCulture | Windows PowerShell: Contains the name of the culture currently in use in the operating system. The culture determines the display format of items such as numbers, currency, and dates. This is the value of the System.Globalization.CultureInfo.CurrentCulture.Name property of the system. To get the System.Globalization.CultureInfo object for the system, use the Get-Culture cmdlet. |
| $PsDebugContext | Windows PowerShell: While debugging, this variable contains information about the debugging environment. Otherwise, it contains $null. As a result, it can be used to indicate whether the debugger has control. When populated, it contains a PsDebugContext object that has Breakpoints and InvocationInfo properties. The InvocationInfo property has several useful properties of its own, including the Location property, which indicates the path of the script that is being debugged. |
| $PsHome | Contains the full path of the installation directory for PowerShell. |
| $PsScriptRoot | Contains the directory from which the script module is being executed. This variable allows scripts to use the module path to access other resources. |
| $PsUICulture | Windows PowerShell: Contains the name of the user interface (UI) culture that is currently in use in the operating system. The UI culture determines which text strings are used for user interface elements, such as menus and messages. This is the value of the System.Globalization.CultureInfo.CurrentUICulture.Name property of the system. To get the System.Globalization.CultureInfo object for the system, use the Get-UICulture cmdlet. |
| $PsVersionTable | Windows PowerShell: Contains a read-only hash table that displays details about the version of Windows PowerShell that is running in the current session. The table includes the following items:  CLRVersion: The version of the common language runtime (CLR) BuildVersion: The build number of the current version PSVersion: The Windows PowerShell version number WSManStackVersion: The version number of the WS-Management stack  PSCompatibleVersions: Versions of Windows PowerShell that are compatible with the current version SerializationVersion The version of the serialization method PSRemotingProtocolVersion: The version of the Windows PowerShell remote management protocol |
| $Pwd | Contains a working location object (§4.5.5) that represents the full path of the current directory. |
| $Sender | Windows PowerShell: Contains the object that generated this event. This variable is populated only within the Action block of an event registration command. The value of this variable can also be found in the Sender property of the System.Management.Automation.PSEventArgs object returned by the cmdlet Get-Event. |
| $ShellID | Windows PowerShell: Contains the identifier of the current shell. |
| $SourceArgs | Windows PowerShell: Contains objects that represent the event arguments of the event that is being processed. This variable is populated only within the Action block of an event registration command. The value of this variable can also be found in the SourceArgs property of the System.Management.Automation.PSEventArgs object returned by the cmdlet Get-Event. |
| $SourceEventArgs | Windows PowerShell: Contains an object that represents the first event argument that derives from EventArgs of the event that is being processed. This variable is populated only within the Action block of an event registration command. The value of this variable can also be found in the SourceArgs property of the System.Management.Automation.PSEventArgs object returned by the cmdlet Get-Event. |
| $switch | An enumerator created for any switch statement. This variable exists only while the switch is executing. The type of an object that represents an enumerator is described in §4.5.16. |
| $this | Windows PowerShell: In a script block that defines a script property or script method, this variable refers to the object that is being extended. |
| $true | Has type bool (§4.2.1) and the constant value True. |

#### Preference variables

Preference variables store user preferences for the session. They are created and initialized by the PowerShell runtime environment. Their values can be read and written in user-written script.

| **Variable** | **Meaning** |
| --- | --- |
| $ConfirmPreference | Type: Confirm-Impact, Default value: High  Indicates an impact level. Cmdlets with an equal or higher impact level can request confirmation before they perform their operation. For example, if $ConfirmPreference is set to Medium, cmdlets with a Medium or High impact level can request confirmation. Requests from cmdlets with a low impact level are suppressed. See §4.2.6.2 for the values allowed. See §12.3.5 for more information. |
| $DebugPreference | Type: Action-Preference, Default value: SilentlyContinue  Determines how the PowerShell environment responds to debugging messages generated by a script or cmdlet. See §4.2.6.1 for the values allowed.  During the execution of any cmdlet, the value of this variable can be overridden by using the common parameter Debug (§13.56). |
| $ErrorActionPreference | Type: Action-Preference, Default value: Continue  Determines how the PowerShell environment responds to a non-terminating error. See §4.2.6.1 for the values allowed.  During the execution of any cmdlet, the value of this variable can be overridden by using the common parameter ErrorAction (§13.56). |
| $ErrorView | Type: string, Default value: "NormalView"  Determines the display format of error messages. The values allowed are:   * "NormalView" – A detailed view designed for most users. Consists of a description of the error, the name of the object involved in the error, and arrows (<<<<) that point to the words in the command that caused the error. * "CategoryView" – A succinct, structured view designed for production environments. The format is {Category}: ({TargetName}:{TargetType}):[{Activity}], {Reason} |
| $FormatEnumerationLimit | Type: int, Default value: 4  Determines how many enumerated items are included in a display. This variable does not affect the underlying objects, just the display.  When the value is less than the number of enumerated items, PowerShell adds an ellipsis to indicate items not shown. |
| $MaximumAliasCount | Type: int, Default value: 4096  Determines how many aliases are permitted in a session. The range of valid values is 1024–32768. |
| $MaximumDriveCount | Type: int, Default value: 4096  Determines how many drives are permitted in a given session. This includes file system drives and data stores exposed by providers and that appear as drives. The range of valid values is 1024–32768. |
| $MaximumErrorCount | Type: int, Default value: 256  The maximum number of error records that can be stored in $Error. For more information, see §3.12. The range of valid values is 256–32768. |
| $MaximumFunctionCount | Type: int, Default value: 4096  Determines how many functions are permitted in a given session. The range of valid values is 1024–32768. |
| $MaximumHistoryCount | Type: int, Default value: 64  Determines how many commands are saved in the command history for the current session. The range of valid values is 1–32768. |
| $MaximumVariableCount | Type: int, Default value: 4096  Determines how many automatic, preference, and user-defined variables are permitted in a given session.  The range of valid values is 1024–32768. |
| $OFS | Type: string, Default value: a space  Contains the Output Field Separator, which separates the elements of an array when the array is converted to a string. |
| $OutputEncoding | Type: object, Default value: ASCIIEncoding  Determines the character encoding method used when text is sent to other applications. The valid values are objects derived from an encoding class, such as ASCIIEncoding, SBCSCodePageEncoding, UTF7Encoding, UTF8Encoding, UTF32Encoding, and UnicodeEncoding. |
| $ProgressPreference | Type: Action-Preference, Default value: Continue  Determines how the runtime responds to progress updates generated by a script or cmdlet. See §4.2.6.1 for the values allowed. |
| $VerbosePreference | Type: Action-Preference, Default value: SilentlyContinue  Determines how the runtime responds to verbose messages generated by a script or cmdlet. Typically, verbose messages describe the actions performed to execute a command. By default, verbose messages are not displayed. See §4.2.6.1 for the values allowed.  During the execution of any cmdlet, the value of this variable can be overridden by using the common parameter Verbose (§13.56). |
| $WarningPreference | Type: Action-Preference, Default value: Continue  Determines how the runtime responds to warning messages generated by a script or cmdlet. See §4.2.6.1 for the values allowed.  During the execution of any cmdlet, the value of this variable can be overridden by using the common parameter WarningAction (§13.56). |
| $WhatIfPreference | Type: bool, Default value: $false  Determines whether WhatIf support is automatically enabled for every command that supports it. When WhatIf is enabled, the cmdlet reports the expected effect of the command, but does not execute the command. The valid values are:   * $false – WhatIf is not automatically enabled. To enable it manually, use the WhatIf parameter of the command. * $true – WhatIf is automatically enabled on any command that supports it. (Users can use the WhatIf switch parameter with a value of $false to disable it manually.) |

### Commands

Syntax:

generic-token:  
generic-token-parts

generic-token-parts:  
generic-token-part  
generic-token-parts generic-token-part

generic-token-part:  
expandable-string-literal  
verbatim-here-string-literal  
variable  
generic-token-char

generic-token-char:  
Any Unicode character except  
 { } ( ) ; , | & $  
 ` (The backtick character U+0060)  
 double-quote-character  
 single-quote-character  
 whitespace  
 new-line-character  
escaped-character

generic-token-with-subexpr-start:  
generic-token-parts $(

### Parameters

Syntax:

command-parameter:  
dash first-parameter-char parameter-chars colonopt

first-parameter-char:  
A Unicode character of classes Lu, Ll, Lt, Lm, or Lo  
\_ (The underscore character U+005F)  
?

parameter-chars:  
parameter-char  
parameter-chars parameter-char

parameter-char:  
Any Unicode character except  
 { } ( ) ; , | & . [  
 colon  
 whitespace  
 new-line-character

colon:  
: (The colon character U+003A)

Description:

When a command is invoked, information may be passed to it via one or more arguments whose values are accessed from within the command through a set of corresponding parameters. The process of matching parameters to arguments is called parameter binding.

There are three kinds of argument:

* Switch parameter (§8.10.4) – This has the form command-parameter where first-parameter-char and parameter-chars together make up the switch name, which corresponds to the name of a parameter (without its leading -) in the command being invoked. If the trailing colon is omitted, the presence of this argument indicates that the corresponding parameter be set to $true. If the trailing colon is present, the argument immediately following must designate a value of type bool, and the corresponding parameter is set to that value. For example, the following invocations are equivalent:  
    
   Set-MyProcess -Strict  
   Set-MyProcess -Strict: $true
* Parameter with argument (§8.10.2) – This has the form command-parameter where first-parameter-char and parameter-chars together make up the parameter name, which corresponds to the name of a parameter (without its leading -) in the command being invoked. There must be no trailing colon. The argument immediately following designates an associated value. For example, given a command Get-Power, which has parameters $base and $exponent, the following invocations are equivalent:  
    
   Get-Power -base 5 -exponent 3  
   Get-Power -exponent 3 -base 5
* Positional argument (§8.10.2) – Arguments and their corresponding parameters inside commands have positions with the first having position zero. The argument in position 0 is bound to the parameter in position 0; the argument in position 1 is bound to the parameter in position 1; and so on. For example, given a command Get-Power, that has parameters $base and $exponent in positions 0 and 1, respectively, the following invokes that command:  
    
   Get-Power 5 3

See §8.2 for details of the special parameter --.

When a command is invoked, a parameter name may be abbreviated; any distinct leading part of the full name may be used, provided that is unambiguous with respect to the names of the other parameters accepted by the same command.

For information about parameter binding see §8.11.

### Literals

Syntax:

literal:  
integer-literal  
real-literal  
string-literal

#### Numeric literals

There are two kinds of numeric literals: integer (§2.3.5.1.1) and real (§2.3.5.1.2). Both can have multiplier suffixes (§2.3.5.1.3).

##### Integer literals

Syntax:

integer-literal:  
decimal-integer-literal  
hexadecimal-integer-literal

decimal-integer-literal:  
decimal-digits numeric-type-suffixopt numeric-multiplieropt

decimal-digits:  
decimal-digit  
decimal-digit decimal-digits

decimal-digit: one of  
0 1 2 3 4 5 6 7 8 9

numeric-type-suffix:  
long-type-suffix  
decimal-type-suffix

hexadecimal-integer-literal:  
0x hexadecimal-digits long-type-suffixopt numeric-multiplieropt

hexadecimal-digits:  
hexadecimal-digit  
hexadecimal-digit decimal-digits

hexadecimal-digit: one of  
0 1 2 3 4 5 6 7 8 9 a b c d e f

long-type-suffix:  
l

numeric-multiplier: one of  
kb mb gb tb pb

Description:

The type of an integer literal is determined by its value, the presence or absence of long-type-suffix, and the presence of a numeric-multiplier (§2.3.5.1.3).

For an integer literal with no long-type-suffix

* If its value can be represented by type int (§4.2.3), that is its type;
* Otherwise, if its value can be represented by type long (§4.2.3), that is its type.
* Otherwise, if its value can be represented by type decimal (§2.3.5.1.2), that is its type.
* Otherwise, it is represented by type double (§2.3.5.1.2).

For an integer literal with long-type-suffix

* If its value can be represented by type long (§4.2.3), that is its type;
* Otherwise, that literal is ill formed.

In the twos-complement representation of integer values, there is one more negative value than there is positive. For the int type, that extra value is ‑2147483648. For the long type, that extra value is ‑9223372036854775808. Even though the token 2147483648 would ordinarily be treated as a literal of type long, if it is preceded immediately by the unary - operator, that operator and literal are treated as a literal of type int having the smallest value. Similarly, even though the token 9223372036854775808 would ordinarily be treated as a real literal of type decimal, if it is immediately preceded by the unary - operator, that operator and literal are treated as a literal of type long having the smallest value.

Some examples of integer literals are 123 (int), 123L (long), and 200000000000 (long).

There is no such thing as an integer literal of type byte.

##### Real literals

Syntax:

real-literal:  
decimal-digits . decimal-digits exponent-partopt decimal-type-suffixopt numeric-multiplieropt  
. decimal-digits exponent-partopt decimal-type-suffixopt numeric-multiplieropt  
decimal-digits exponent-part decimal-type-suffixopt numeric-multiplieropt

exponent-part:  
e signopt decimal-digits

sign: one of  
+  
*dash*

decimal-type-suffix:  
d

numeric-multiplier: one of  
kb mb gb tb pb

dash:  
- (U+002D)  
EnDash character (U+2013)  
EmDash character (U+2014)  
Horizontal bar character (U+2015)

Description:

A real literal may contain a numeric-multiplier (§2.3.5.1.3).

There are two kinds of real literal: double and decimal. These are indicated by the absence or presence, respectively, of decimal-type-suffix. (There is no such thing as a float real literal.)

A double real literal has type double (§4.2.4.1). A decimal real literal has type decimal (§4.2.4.2). Trailing zeros in the fraction part of a decimal real literal are significant.

If the value of exponent-part's decimal-digits in a double real literal is less than the minimum supported, the value of that double real literal is 0. If the value of exponent-part's decimal-digits in a decimal real literal is less than the minimum supported, that literal is ill formed. If the value of exponent-part's decimal-digits in a double or decimal real literal is greater than the maximum supported, that literal is ill formed.

Some examples of double real literals are 1., 1.23, .45e35, 32.e+12, and 123.456E-231.

Some examples of decimal real literals are 1d (which has scale 0), 1.20d (which has scale 2), 1.23450e1d (i.e., 12.3450, which has scale 4), 1.2345e3d (i.e., 1234.5, which has scale 1), 1.2345e-1d (i.e., 0.12345, which has scale 5), and 1.2345e-3d (i.e., 0.0012345, which has scale 7).

[Note: Because a double real literal need not have a fraction or exponent part, the grouping parentheses in (123).M are needed to ensure that the property or method M is being selected for the integer object whose value is 123. Without those parentheses, the real literal would be ill formed. end note]

[Note: Although PowerShell does not provide literals for infinities and NaNs, double real literal-like equivalents can be obtained from the static read-only properties PositiveInfinity, NegativeInfinity, and NaN of the types float and double (§4.2.4.1), end note]

The grammar permits what starts out as a double real literal to have an l or L type suffix. Such a token is really an integer literal whose value is represented by type long. [Note: This feature has been retained for backwards compatibility with earlier versions of PowerShell. However, programmers are discouraged from using integer literals of this form as they can easily obscure the literal's actual value. For example, 1.2L has value 1, 1.2345e1L has value 12, and 1.2345e-5L has value 0, none of which is immediately obvious. end note]

##### Multiplier suffixes

Syntax:

numeric-multiplier: one of  
kb mb gb tb pb

Description:

For convenience, integer and real literals can contain a numeric-multiplier, which indicates one of a set of commonly used powers of 10. numeric-multiplier can be written in any combination of upper- or lowercase letters.

| **Multiplier** | **Meaning** | **Example** |
| --- | --- | --- |
| kb | kilobyte (1024) | 1kb ≡ 1024 |
| mb | megabyte (1024 x 1024) | 1.30Dmb ≡ 1363148.80 |
| gb | gigabyte (1024 x 1024 x 1024) | 0x10Gb ≡ 17179869184 |
| tb | terabyte (1024 x 1024 x 1024 x 1024) | 1.4e23tb ≡ 1.5393162788864E+35 |
| pb | petabyte (1024 x 1024 x 1024 x 1024 x 1024) | 0x12Lpb ≡ 20266198323167232 |

#### String literals

Syntax:

string-literal:  
expandable-string-literal  
expandable-here-string-literal  
verbatim-string-literal  
verbatim-here-string-literal

expandable-string-literal:  
double-quote-character expandable-string-charactersopt dollarsopt double-quote-character

double-quote-character:  
" (U+0022)  
Left double quotation mark (U+201C)  
Right double quotation mark (U+201D)  
Double low-9 quotation mark (U+201E)

expandable-string-characters:  
expandable-string-part  
expandable-string-characters expandable-string-part

expandable-string-part:  
Any Unicode character except  
 $  
 double-quote-character  
 ` (The backtick character U+0060)  
braced-variable  
$ Any Unicode character except  
 (  
 {  
 double-quote-character  
 ` (The backtick character U+0060)  
$ *escaped-character*  
*escaped-character  
double-quote-character double-quote-character*

*dollars:*$ *dollars* $

expandable-here-string-literal:  
@ double-quote-character whitespaceopt new-line-character  
 expandable-here-string-charactersopt new-line-character double-quote-character @

expandable-here-string-characters:  
expandable-here-string-part  
expandable-here-string-characters expandable-here-string-part

expandable-here-string-part:  
Any Unicode character except  
 $  
 new-line-character  
braced-variable  
$ Any Unicode character except  
 (  
 new-line-character  
$ new-line-character Any Unicode character except double-quote-char  
$ new-line-character double-quote-char Any Unicode character except @  
new-line-character Any Unicode character except double-quote-char  
new-line-character double-quote-char Any Unicode character except @

expandable-string-with-subexpr-start:  
double-quote-character expandable-string-charsopt $(

expandable-string-with-subexpr-end:  
double-quote-char

expandable-here-string-with-subexpr-start:  
@ double-quote-character whitespaceopt new-line-character  
 expandable-here-string-charsopt $(

expandable-here-string-with-subexpr-end:  
new-line-character double-quote-character @

verbatim-string-literal:  
single-quote-character verbatim-string-charactersopt single-quote-char

single-quote-character:  
' (U+0027)  
Left single quotation mark (U+2018)  
Right single quotation mark (U+2019)  
Single low-9 quotation mark (U+201A)  
Single high-reversed-9 quotation mark (U+201B)

verbatim-string-characters:  
verbatim-string-part  
verbatim-string-characters verbatim-string-part

verbatim-string-part:  
Any Unicode character except single-quote-character  
single-quote-character single-quote-character

verbatim-here-string-literal:  
@ single-quote-character whitespaceopt new-line-character  
 verbatim-here-string-charactersopt new-line-character single-quote-character @

verbatim-*here-string-characters:  
verbatim-here-string-part  
verbatim-here-string-characters verbatim-here-string-part*

*verbatim-here-string-part:*Any Unicode character except new-line-character  
new-line-character Any Unicode character except single-quote-character  
new-line-character single-quote-character Any Unicode character except @

Description:

There are four kinds of string literals:

* verbatim-string-literal (single-line single-quoted), which is a sequence of zero or more characters delimited by a pair of single-quote-characters. Examples are '' and 'red'.
* expandable-string-literal (single-line double-quoted), which is a sequence of zero or more characters delimited by a pair of double-quote-characters. Examples are "" and "red".
* verbatim-here-string-literal (multi-line single-quoted), which is a sequence of zero or more characters delimited by the character pairs @single-quote-character and single-quote-character@, respectively, all contained on two or more source lines. Examples are:  
    
  @'  
  '@  
    
  @'  
  line 1  
  '@  
    
  @'  
  line 1  
  line 2  
  '@
* expandable-here-string-literal (multi-line double-quoted), which is a sequence of zero or more characters delimited by the character pairs @double-quote-character and double-quote-character@, respectively, all contained on two or more source lines. Examples are:  
    
  @"  
  "@  
    
  @"  
  line 1  
  "@  
    
  @"  
  line 1  
  line 2  
  "@

For verbatim-here-string-literals and expandable-here-string-literals, except for white space (which is ignored) no characters may follow on the same source line as the opening delimiter-character pair, and no characters may precede on the same source line as the closing delimiter character pair.

The body of a verbatim-here-string-literal or an expandable-here-string-literal begins at the start of the first source line following the opening delimiter, and ends at the end of the last source line preceding the closing delimiter. The body may be empty. The line terminator on the last source line preceding the closing delimiter is not part of that literal's body.

A literal of any of these kinds has type string (§4.3.1).

The character used to delimit a verbatim-string-literal or expandable-string-literal can be contained in such a string literal by writing that character twice, in succession. For example, 'What''s the time?' and "I said, ""Hello"".". However, a single-quote-character has no special meaning inside an expandable-string-literal, and a double-quote-character has no special meaning inside a verbatim-string-literal.

An expandable-string-literal and an expandable-here-string-literal may contain escaped-characters (§2.3.7). For example, when the following string literal is written to the pipeline, the result is as shown below:

"column1`tcolumn2`nsecond line, `"Hello`", ```Q`5`!"

column1<horizontal-tab>column2<new-line>  
second line, "Hello", `Q5!

If an expandable-string-literal or expandable-here-string-literal contains the name of a variable, unless that name is preceded immediately by an escape character, it is replaced by the string representation of that variable's value (§6.7). This is known as variable substitution. [Note: If the variable name is part of some larger expression, only the variable name is replaced. For example, if $a is an array containing the elements 100 and 200, ">$a.Length<" results in >100 200.Length< while ">$($a.Length)<" results in >2<. See sub-expression expansion below. end note]

For example, the source code

$count = 10  
"The value of `$count is $count"

results in the expandable-string-literal

The value of $count is 10.

Consider the following:

$a = "red","blue"  
"`$a[0] is $a[0], `$a[0] is $($a[0])" # second [0] is taken literally

The result is

$a[0] is red blue[0], $a[0] is red

expandable-string-literals and expandable-here-string-literals also support a kind of substitution called sub-expression expansion, by treating text of the form $( … ) as a sub-expression (§7.1.6). Such text is replaced by the string representation of that expression's value (§6.8). Any white space used to separate tokens within sub-expression's statement-list is ignored as far as the result string's construction is concerned.

The examples,

$count = 10  
"$count + 5 is $($count + 5)"  
"$count + 5 is `$($count + 5)"  
"$count + 5 is `$(`$count + 5)"

result in the following expandable-string-literals:

10 + 5 is 15  
10 + 5 is $(10 + 5)  
10 + 5 is $($count + 5)

The following source,

$i = 5; $j = 10; $k = 15  
"`$i, `$j, and `$k have the values $( $i; $j; $k )"

results in the following expandable-string-literal:

$i, $j, and $k have the values 5 10 15

These four lines could have been written more succinctly as follows:

"`$i, `$j, and `$k have the values $(($i = 5); ($j = 10); ($k = 15))"

In the following example,

"First 10 squares: $(for ($i = 1; $i -le 10; ++$i) { "$i $($i\*$i) " })"

the resulting expandable-string-literal is as follows:

First 10 squares: 1 1 2 4 3 9 4 16 5 25 6 36 7 49 8 64 9 81 10 100

As shown, a sub-expression can contain string literals having both variable substitution and sub-expression expansion. Note also that the inner expandable-string-literal's delimiters need not be escaped; the fact that they are inside a sub-expression means they cannot be terminators for the outer expandable-string-literal.

An expandable-string-literal or expandable-here-string-literal containing a variable substitution or sub-expression expansion is evaluated each time that literal is used; for example,

$a = 10  
$s1 = "`$a = $($a; ++$a)"   
"`$s1 = >$s1<"  
$s2 = "`$a = $($a; ++$a)"  
"`$s2 = >$s2<"  
$s2 = $s1  
"`$s2 = >$s2<"

which results in the following expandable-string-literals:

$s1 = >$a = 10<  
$s2 = >$a = 11<  
$s2 = >$a = 10<

The contents of a verbatim-here-string-literal are taken verbatim, including any leading or trailing white space within the body. As such, embedded single-quote-characters need not be doubled-up, and there is no substitution or expansion. For example,

$lit = @'  
That's it!  
2 \* 3 = $(2\*3)  
'@

which results in the literal

That's it!  
2 \* 3 = $(2\*3)

The contents of an expandable-here-string-literal are subject to substitution and expansion, but any leading or trailing white space within the body but outside any sub-expressions is taken verbatim, and embedded double-quote-characters need not be doubled-up. For example,

$lit = @"  
That's it!  
2 \* 3 = $(2\*3)  
"@

which results in the following literal when expanded:

That's it!  
2 \* 3 = 6

For both verbatim-here-string-literals and expandable-here-string-literals, each line terminator within the body is represented in the resulting literal in an implementation-defined manner. For example, in

$lit = @"  
abc  
 xyz  
"@

the second line of the body has two leading spaces, and the first and second lines of the body have line terminators; however, the terminator for the second line of the body is not part of that body. The resulting literal is equivalent to "abc<implementation-defined character sequence>  xyz".

[Note: To aid readability of source, long string literals can be broken across multiple source lines without line terminators being inserted. This is done by writing each part as a separate literal and concatenating the parts with the + operator (§7.7.2). This operator allows its operands to designate any of the four kinds of string literal. end note]

[Note: Although there is no such thing as a character literal per se, the same effect can be achieved by accessing the first character in a 1-character string, as follows: [char]"A" or "A"[0]. end note]

Windows PowerShell: For both verbatim-here-string-literals and expandable-here-string-literals, each line terminator within the body is represented exactly as it was provided.

#### Null literal

See the automatic variable $null (§2.3.2.2).

#### Boolean literals

See the automatic variables $false and $true (§2.3.2.2).

#### Array literals

PowerShell allows expressions of array type (§9) to be written using the unary comma operator (§7.2.1), array-expression (§7.1.7), the binary comma operator (§7.3), and the range operator (§7.4).

#### Hash literals

PowerShell allows expressions of type Hashtable (§10) to be written using a hash-literal-expression (§7.1.9)

#### Type names

Syntax:

type-name:  
type-identifier  
type-name . type-identifier

type-identifier:  
type-characters

type-characters:  
type-character  
type-characters type-character

type-character:  
A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nd  
\_ (The underscore character U+005F)

array-type-name:  
type-name [

generic-type-name:  
type-name [

### Operators and punctuators

Syntax:

operator-or-punctuator: one of  
{ } [ ] ( ) @( @{ $( ;  
&& || & | , ++ .. :: .  
! \* / % + 2>&1 1>&2  
dash dash dash  
dash and dash band dash bnot  
dash bor dash bxor dash not  
dash or dash xor  
assignment-operator   
file-redirection-operator  
comparison-operator  
format-operator

assignment-operator: one of  
= dash = += \*= /= %=

file-redirection-operator: one of  
>> > < 2>> 2>

comparison-operator: one of  
dash as dash ccontains dash ceq  
dash cge dash cgt dash cle  
dash clike dash clt dash cmatch  
dash cne dash cnotcontains dash cnotlike  
dash cnotmatch dash contains dash creplace  
dash csplit dash eq dash ge  
dash gt dash icontains dash ieq  
dash ige dash igt dash ile  
dash ilike dash ilt dash imatch  
dash ine dash inotcontains dash inotlike  
dash inotmatch dash ireplace dash is  
dash isnot dash isplit dash join  
dash le dash like dash lt  
dash match dash ne dash notcontains  
dash notlike dash notmatch dash replace  
dash split

format-operator:  
dash f

Description:

&& and || are reserved for future use.

The name following dash in an operator is reserved for that purpose only in an operator context.

An operator that begins with dash must not have any white space between that dash and the token that follows it.

### Escaped characters

Syntax:

escaped-character:  
` (The backtick character U+0060) followed by any Unicode character

Description:

An escaped character is a way to assign a special interpretation to a character by giving it a prefix Backtick character (U+0060). The following table shows the meaning of each escaped-character:

| Escaped Character | Meaning |
| --- | --- |
| `a | Alert (U+0007) |
| `b | Backspace (U+0008) |
| `f | Form-feed (U+000C) |
| `n | New-line (U+000A) |
| `r | Carriage return (U+000D) |
| `t | Horizontal tab (U+0009) |
| `v | Vertical tab (U+0009) |
| `' | Single quote (U+0027) |
| `" | Double quote (U+0022) |
| `` | Backtick (U+0060) |
| `0 | NUL (U+0000) |
| `x | If x is a character other than those characters shown above, the backtick character is ignored and x is taken literally. |

The implication of the final entry in the table above is that spaces that would otherwise separate tokens can be made part of a token instead. For example, a file name containing a space can be written as Test` Data.txt (as well as 'Test Data.txt' or "Test Data.txt").

# Basic concepts

## Providers and drives

A provider allows access to data and components that would not otherwise be easily accessible at the command line. The data is presented in a consistent format that resembles a file system drive.

The data that a provider exposes appears on a drive, and the data is accessed via a path just like with a disk drive. Built-in cmdlets for each provider manage the data on the provider drive.

PowerShell includes the following set of built-in providers to access the different types of data stores:

| **Provider** | **Drive Name** | **Description** | **Ref.** |
| --- | --- | --- | --- |
| Alias | Alias: | PowerShell aliases | §3.1.1 |
| Environment | Env: | Environment variables | §3.1.2 |
| FileSystem | A:, B:, C:, … | Disk drives, directories, and files | §3.1.3 |
| Function | Function: | PowerShell functions | §3.1.4 |
| Variable | Variable: | PowerShell variables | §3.1.5 |

Windows PowerShell:

| **Provider** | **Drive Name** | **Description** |
| --- | --- | --- |
| Certificate | Cert: | x509 certificates for digital signatures |
| Registry | HKLM: (HKEY\_LOCAL\_MACHINE), HKCU: (HKEY\_CURRENT\_USER) | Windows registry |
| WSMan | WSMan: | WS-Management configuration information |

The following cmdlets deal with providers and drives:

* Get-PSProvider: Gets information about one or more providers (see §13.25)
* Get-PSDrive: Gets information about one or more drives (see §13.24)

The type of an object that represents a provider is described in §4.5.1. The type of an object that represents a drive is described in §4.5.2.

### Aliases

An alias is an alternate name for a command. A command can have multiple aliases, and the original name and all of its aliases can be used interchangeably. An alias can be reassigned. An alias is an item (§3.3).

An alias can be assigned to another alias; however, the new alias is not an alias of the original command.

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

Some aliases are built in to PowerShell. (For those built-in cmdlets having aliases, those aliases follow their cmdlets name in the section heading of §13.)

The following cmdlets deal with aliases:

* New-Alias: Creates an alias (see §13.33)
* Set-Alias: Creates or changes one or more aliases (see §13.46)
* Get-Alias: Gets information about one or more aliases (see §13.13)
* Export-Alias: Exports one or more aliases to a file (see §13.10)

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

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).

### 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).

### 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).

### 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).

### 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 (see §13.37)
* Set-Variable: Creates or changes the characteristics of one or more variables (see §13.50)
* Get-Variable: Gets information about one or more variables (see §13.26)
* Clear-Variable: Deletes the value of one or more variables (see §13.5)
* Remove-Variable: Deletes one or more variables (see §13.42)

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).

## 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 (see §13.49)
* Get-Location: Determines the current working location for the specified drive(s), or the working locations for the specified stack(s) (see §13.21)
* Push-Location: Saves the current working location on the top of a specified stack of locations (see §13.39)
* Pop-Location: Restores the current working location from the top of a specified stack of locations (see §13.38)

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

## 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 (see §13.34)
* Set-Item: Changes the value of one or more items (see §13.48)
* Get-Item: Gets the items at the specified location (see §13.17)
* Get-ChildItem: Gets the items and child items at the specified location (see §13.14)
* Copy-Item: Copies one or more items from one location to another (see §13.9)
* Move-Item: Moves one or more items from one location to another (see §13.32)
* Rename-Item: Renames an item (see §13.43)
* Invoke-Item: Performs the default action on one or more items (see §13.29)
* Clear-Item: Deletes the contents of one or more items, but does not delete the items (see §13.4)
* Remove-Item: Deletes the specified items (see §13.40)

The following cmdlets deal with the content of items:

* Get-Content: Gets the content of the item (see §13.16)
* Add-Content: Adds content to the specified items (see §13.1)
* Set-Content: Writes or replaces the content in an item (see §13.47)
* Clear-Content: Deletes the contents of an item (see §13.3)

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.

## 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:

path:  
provideropt   driveopt   containersopt   item

provider:  
moduleopt   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 (see §13.8)
* Join-Path: Combines a path and a child path into a single path (see §13.30)
* Resolve-Path: Resolves the wildcard characters in a path (see §13.44)
* Split-Path: Returns the specified part of a path (see §13.52)
* Test-Path: Determines whether the elements of a path exist or if a path is well formed (see §13.54)

Some cmdlets (such as Add-Content (§13.1) and Copy-Item (§13.9)) use file filters. A file filter is a mechanism for specifying the criteria for selecting from a set of paths.

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

## Scopes

### 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 $x and $y created in the script is the body of that script, including the function defined inside it. Function Get-Power 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.

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

### 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.)

### Variable name scope

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

variable-scope:  
global:  
local:  
private:  
script:  
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 |
| script | Nearest ancestor script file's scope | Nearest ancestor script file's scope | Nearest ancestor script file's scope |
| local | Global/Script/Local scope | Local scope | Local scope |
| 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 Scope, and the parameters Option Private and Option AllScope for more information.

### 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).

### 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

### 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 (§13.28) allows exported names to have increased visibility.

## 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.

## Method overloads and call resolution

### 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 Abs, 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(decimal)  
Abs(float)  
Abs(double)  
Abs(int)  
Abs(long)

Windows PowerShell: There are two other Abs methods: Abs(SByte) and 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

Windows PowerShell: 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.

### Method overload resolution

Given a method call (§7.1.3) having a list of argument expressions, and a set of candidate methods (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.

### 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.

Windows PowerShell: If the argument type is ref, the corresponding parameter could be out instead of 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.

### Better method

Given an argument list A with a set of argument expressions { E1, E2, …, EN } and two application methods MP and MQ with parameter types { P1, P2, …, PN } and { Q1, Q2, …, QN }, MP is defined to be a better method than MQ if the cumulative ranking of conversions for MP is better than that for MQ.

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 E1 worth N, E2 worth N-1, down to EN worth 1.  If the conversion from EX to PX is better than that from EX to QX, the MP accumulates N-X+1; otherwise, MQ accumulates N-X+1.  If MP and MQ have the same value, then the following tie breaking rules are used, applied in order:

* The cumulative ranking of conversions between parameter types (ignoring argument types) is computed in a manner similar to the previous ranking, so P1 is compared against Q1, P2 against Q2, …, and PN against QN.  The comparison is skipped if the argument was $null, or if the parameter types are not numeric types.  The comparison is also skipped if the argument conversion EX loses information when converted to PX but does not lose information when converted to QX, or vice versa.  If the parameter conversion types are compared, then if the conversion from PX to QX is better than that from QX to PX, the MP accumulates N-X+1; otherwise, MQ 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.
* 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.

### Better conversion

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

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

* T1[] to T2[] where no assignable conversion between T1 and T2 exists
* T to string where T is any type
* T1 to T2 where T1 or T2 define a custom conversion in an implementation-defined manner
* T1 to T2 where T1 implements IConvertible
* T1 to T2 where T1 or T2 implements the method T2 op\_Implicit(T1)
* T1 to T2 where T1 or T2 implements the method T2 op\_Explicit(T1)
* T1 to T2 where T2 implements a constructor taking a single argument of type T1
* Either of the following conversions:
* string to T where T implements a static method T Parse(string) or T Parse(string, IFormatProvider)
* T1 to T2 where T2 is any enum and T1 is either string or a collection of objects that can be converted to string
* T to PSObject where T is any type
* Any of the following conversions: Language
* T to bool where T is any numeric type
* string to T where T is regex, wmisearcher, wmi, wmiclass, adsi, adsisearcher, or type
* T to bool
* T1 to Nullable[T2] where a conversion from T1 to T2 exists
* T to void
* T1[] to T2[] where an assignable conversion between T1 and T2 exists
* T1 to T2[] where T1 is a collection
* IDictionary to Hashtable
* T to ref
* T to xml
* scriptblock to delegate
* T1 to T2 where T1 is an integer type and T2 is an enum
* $null to T where T is any value type
* $null to T where T is any reference type
* Any of the following conversions:
* byte to T where T is SByte
* UInt16 to T where T is SByte, byte, or Int16
* Int16 to T where T is SByte or byte
* UInt32 to T where T is SByte, byte, Int16, UInt16, or int
* int to T where T is SByte, byte, Int16, or UInt16
* UInt64 to T where T is SByte, byte, Int16, UInt16, int, UInt32, or long
* long to T where T is SByte, byte, Int16, UInt16, int, or UInt32
* float to T where T is any integer type or decimal
* double to T where T is any integer type or decimal
* decimal to T where T is any integer type
* Any of the following conversions:
* SByte to T where T is byte, uint6, UInt32, or UInt64
* Int16 to T where T is UInt16, UInt32, or UInt64
* int to T where T is UInt32 or UInt64
* long to UInt64
* decimal to T where T is float or double
* Any of the following conversions:
* T to string where T is any numeric type
* T to char where T is any numeric type
* string to T where T is any numeric type
* Any of the following conversions, these conversion are considered an assignable conversions:
* byte to T where T is Int16, UInt16, int, UInt32, long, UInt64, single, double, or decimal
* SByte to T where T is Int16, UInt16, int, UInt32, long, UInt64, single, double, or decimal
* UInt16 to T where T is int, UInt32, long, or UInt64, single, double, or decimal
* Int16 to T where T is int, UInt32, long, or UInt64, single, double, or decimal
* UInt32 to T where T is long, or UInt64, single, double, or decimal
* int to T where T is long, UInt64, single, double, or decimal
* single to double
* T1 to T2 where T2 is a base class or interface of T1.  This conversion is considered an assignable conversion.
* string to char[]
* T to T – This conversion is considered an assignable conversion.

For each conversion of the form T1 to T2[] where T1 is not an array and no other conversion applies, if there is a conversion from T1 to T2, the rank of the conversion is worse than the conversion from T1 to T2, but better than any conversion ranked less than the conversion from T1 to T2

## 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.

## 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 int, double, long[], and Hashtable.

Windows PowerShell: 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 System.. 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.

## 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.

## 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.

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

$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 (§13.56), which allows a user-defined variable to be specified to hold the collection.

## 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 Get-ChildItem, Sort-Object, and Process-File, 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  
Get-ChildItem e:\\*.txt | Sort-Object -CaseSensitive | Process-File  
 >results.txt

In the first case, Get-ChildItem 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.

In the second case, Get-ChildItem creates a collection of names of the files in the directory specified, using the argument e:\\*.txt. That collection is written to the command Sort-Object, 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 Process-File, which performs some (unknown) processing. The output from that command is then redirected to the 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.

## 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.

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

Modules are discussed in detail in §11.

## 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 [\*]. |
| ? | 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. |

[Note: More information can be found in, "The Open Group Base Specifications: Pattern Matching", IEEE Std 1003.1, 2004 Edition. <http://www.opengroup.org/onlinepubs/000095399/utilities/xcu_chap02.html#tag_02_13_01>. However, in PowerShell, the escape character is backtick, not backslash. end note]

## 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. |

[Note: More information can be found in, "The Open Group Base Specifications: Regular Expressions", IEEE Std 1003.1, 2004 Edition. <http://www.opengroup.org/onlinepubs/000095399/basedefs/xbd_chap09.html>. end note]

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 [\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Nd}\p{Pc}]. If ECMAScript-compliant behavior is specified with the ECMAScript option, \w is equivalent to [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 \p{Nd} for Unicode and [0-9] for non-Unicode behavior. |
| \D | Matches any non-digit. Equivalent to \P{Nd} for Unicode and [^0-9] for non-Unicode behavior. |

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

| Element | Description |
| --- | --- |
| \* | Specifies zero or more matches; for example, \w\* or (abc)\*. Equivalent to {0,}. |
| + | Matches repeating instances of the preceding characters. |
| ? | Specifies zero or one matches; for example, \w? or (abc)?. Equivalent to {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. |

# Types

In PowerShell, each value has a type, and types fall into one of two main categories: value types and reference types. Consider the type int, which is typical of value types. A value of type int is completely self-contained; all the bits needed to represent that value are stored in that value, and every bit pattern in that value represents a valid value for its type. Now, consider the array type int[], which is typical of reference types. A so-called value of an array type can hold either a reference to an object that actually contains the array elements, or the null reference whose value is $null. The important distinction between the two type categories is best demonstrated by the differences in their semantics during assignment. For example,

$i = 100 # $i designates an int value 100  
$j = $i # $j designates an int value 100, which is a copy

$a = 10,20,30 # $a designates an object[], Length 3, value 10,20,30  
$b = $a # $b designates exactly the same array as does $a, not a copy  
$a[1] = 50 # element 1 (which has a value type) is changed from 20 to 50   
$b[1] # $b refers to the same array as $a, so $b[1] is 50

As we can see, the assignment of a reference type value involves a shallow copy; that is, a copy of the reference to the object rather than its actual value. In contrast, a deep copy requires making a copy of the object as well.

A numeric type is one that allows representation of integer or fractional values, and that supports arithmetic operations on those values. The set of numerical types includes the integer (§4.2.3) and real number (§4.2.4) types, but does not include bool (§4.2.1) or char (§4.2.2). An implementation may provide other numeric types (such as signed byte, unsigned integer, and integers of other sizes).

A collection is a group of one or more related items, which need not have the same type. Examples of collection types are arrays, stacks, queues, lists, and hash tables. A program can enumerate (or iterate) over the elements in a collection, getting access to each element one at a time. Common ways to do this are with the foreach statement (§8.4.4) and the ForEach-Object cmdlet (§13.12). The type of an object that represents an enumerator is described in §4.5.16.

In this chapter, there are tables that list the accessible members for a given type. For methods, the "Type" is written with the following form: returnType/argumentTypeList. If the argument type list is too long to fit in that column, it is shown in the "Purpose" column instead.

Windows PowerShell: Other integer types are SByte, Int16, UInt16, UInt32, and UInt64, all in the namespace System.

Windows PowerShell: Many collection classes are defined as part of the System.Collections or System.Collections.Generic namespaces. Most collection classes implement the interfaces ICollection, IComparer, IEnumerable, IList, IDictionary, and IDictionaryEnumerator and their generic equivalents.

## Special types

### The void type

This type cannot be instantiated. It provides a means to discard a value explicitly using the cast operator (§7.2.9).

### The null type

The null type has one instance, the automatic variable $null (§2.3.2.2), also known as the null value. This value provides a means for expressing "nothingness" in reference contexts. The characteristics of this type are unspecified.

### The object type

Every type in PowerShell except the null type (§4.1.2) is derived directly or indirectly from the type object, so object is the ultimate base type of all non-null types. A variable constrained (§5.3) to type object is really not constrained at all, as it can contain a value of any type.

## Value types

### Boolean

The Boolean type is bool. There are only two values of this type, False and True, represented by the automatic variables $false and $true, respectively (§2.3.2.2).

Windows PowerShell: bool maps to System.Boolean.

### Character

A character value has type char, which is capable of storing any UTF-16-encoded 16-bit Unicode code point.

The type char has the following accessible members:

| **Member** | **Member Kind** | **Type** | **Purpose** |
| --- | --- | --- | --- |
| MaxValue | Static property (read-only) | char | The largest possible value of type char |
| MinValue | Static property (read-only) | char | The smallest possible value of type char |
| IsControl | Static method | bool/char | Tests if the character is a control character |
| IsDigit | Static method | bool/char | Tests if the character is a decimal digit |
| IsLetter | Static method | bool/char | Tests if the character is an alphabetic letter |
| IsLetterOrDigit | Static method | bool/char | Tests if the character is a decimal digit or alphabetic letter |
| IsLower | Static method | bool/char | Tests if the character is a lowercase alphabetic letter |
| IsPunctuation | Static method | bool/char | Tests if the character is a punctuation mark |
| IsUpper | Static method | bool/char | Tests if the character is an uppercase alphabetic letter |
| IsWhiteSpace | Static method | bool/char | Tests if the character is a white space character. |
| ToLower | Static method | char/string | Converts the character to lowercase |
| ToUpper | Static method | char/string | Converts the character to uppercase |

Windows PowerShell: char maps to System.Char.

### Integer

There are two signed integer types, both of use two's-complement representation for negative values:

* Type int, which uses 32 bits giving it a range of -2147483648 to +2147483647, inclusive.
* Type long, which uses 64 bits giving it a range of -9223372036854775808 to +9223372036854775807, inclusive.

Type int has the following accessible members:

| **Member** | **Member Kind** | **Type** | **Purpose** |
| --- | --- | --- | --- |
| MaxValue | Static property (read-only) | int | The largest possible value of type int |
| MinValue | Static property (read-only) | int | The smallest possible value of type int |

Type long has the following accessible members:

| **Member** | **Member Kind** | **Type** | **Purpose** |
| --- | --- | --- | --- |
| MaxValue | Static property (read-only) | long | The largest possible value of type long |
| MinValue | Static property (read-only) | long | The smallest possible value of type long |

There is one unsigned integer type:

* Type byte, which uses 8 bits giving it a range of 0 to 255, inclusive.

Type byte has the following accessible members:

| **Member** | **Member Kind** | **Type** | **Purpose** |
| --- | --- | --- | --- |
| MaxValue | Static property (read-only) | byte | The largest possible value of type byte |
| MinValue | Static property (read-only) | byte | The smallest possible value of type byte |

Windows PowerShell: byte, int, and long map to System.Byte, System.Int32, and System.Int64, respectively.

### Real number

#### float and double

There are two real (or floating-point) types:

* Type float uses the 32-bit IEEE single-precision representation.
* Type double uses the 64-bit IEEE double-precision representation.

A third type name, single, is a synonym for type float; float is used throughout this specification.

Although the size and representation of the types float and double are defined by this specification, an implementation may use extended precision for intermediate results.

Type float has the following accessible members:

| **Member** | **Member Kind** | **Type** | **Purpose** |
| --- | --- | --- | --- |
| MaxValue | Static property (read-only) | float | The largest possible value of type float |
| MinValue | Static property (read-only) | float | The smallest possible value of type float |
| NaN | Static property (read-only) | float | The constant value Not-a-Number |
| NegativeInfinity | Static property (read-only) | float | The constant value negative infinity |
| PositiveInfinity | Static property (read-only) | float | The constant value positive infinity |

Type double has the following accessible members:

| **Member** | **Member Kind** | **Type** | **Purpose** |
| --- | --- | --- | --- |
| MaxValue | Static property (read-only) | double | The largest possible value of type double |
| MinValue | Static property (read-only) | double | The smallest possible value of type double |
| NaN | Static property (read-only) | double | The constant value Not-a-Number |
| NegativeInfinity | Static property (read-only) | double | The constant value negative infinity |
| PositiveInfinity | Static property (read-only) | double | The constant value positive infinity |

Windows PowerShell: float and double map to System.Single and System.Double, respectively.

#### decimal

Type decimal uses a 128-bit representation. At a minimum it must support a scale s such that 0 <= s <= at least 28, and a value range -79228162514264337593543950335 to 79228162514264337593543950335. The actual representation of decimal is implementation defined.

Type decimal has the following accessible members:

| **Member** | **Member Kind** | **Type** | **Purpose** |
| --- | --- | --- | --- |
| MaxValue | Static property (read-only) | decimal | The largest possible value of type decimal |
| MinValue | Static property (read-only) | decimal | The smallest possible value of type decimal |

[Note: Decimal real numbers have a characteristic called scale, which represents the number of digits to the right of the decimal point. For example, the value 2.340 has a scale of 3 where trailing zeros are significant. When two decimal real numbers are added or subtracted, the scale of the result is the larger of the two scales. For example, 1.0 + 2.000 is 3.000, while 5.0 - 2.00 is 3.00. When two decimal real numbers are multiplied, the scale of the result is the sum of the two scales. For example, 1.0 \* 2.000 is 2.0000. When two decimal real numbers are divided, the scale of the result is the scale of the first less the scale of the second. For example, 4.00000/2.000 is 2.00. However, a scale cannot be less than that needed to preserve the correct result. For example, 3.000/2.000, 3.00/2.000, 3.0/2.000, and 3/2 are all 1.5. end note]

Windows PowerShell: decimal maps to System.Decimal. The representation of decimal is as follows: When considered as an array of four ints it contains the following elements: Index 0 (bits 0‑31) contains the low-order 32 bits of the decimal's coefficient. Index 1 (bits 32‑63) contains the middle 32 bits of the decimal's coefficient. Index 2 (bits 64‑95) contains the high-order 32 bits of the decimal's coefficient. Index 3 (bits 96‑127) contains the sign bit and scale, as follows: bits 0–15 are zero, bits 16‑23 contains the scale as a value 0–28, bits 24‑30 are zero, and bit 31 is the sign (0 for positive, 1 for negative).

### The switch type

This type is used to constrain the type of a parameter in a command (§8.10.4). If an argument having the corresponding parameter name is present the parameter tests $true; otherwise, it tests $false.

Windows PowerShell: switch maps to System.Management.Automation.SwitchParameter.

### Enumeration types

An enumeration type is one that defines a set of named constants representing all the possible values that can be assigned to an object of that enumeration type. In some cases, the set of values are such that only one value can be represented at a time. In other cases, the set of values are distinct powers of two, and by using the -bor operator (§7.9), multiple values can be encoded in the same object.

The PowerShell environment provides a number of enumeration types, as described in the following sections.

#### Action-Preference type

This implementation-defined type has the following mutually exclusive-valued accessible members:

| **Member** | **Member Kind** | **Purpose** |
| --- | --- | --- |
| Continue | Enumeration constant | The PowerShell runtime will continue processing and notify the user that an action has occurred. |
| Inquire | Enumeration constant | The PowerShell runtime will stop processing and ask the user how it should proceed. |
| SilentlyContinue | Enumeration constant | The PowerShell runtime will continue processing without notifying the user that an action has occurred. |
| Stop | Enumeration constant | The PowerShell runtime will stop processing when an action occurs. |

Windows PowerShell: This type is System.Management.Automation.ActionPreference.

#### Confirm-Impact type

This implementation-defined type has the following mutually exclusive-valued accessible members:

| **Member** | **Member Kind** | **Purpose** |
| --- | --- | --- |
| High | Enumeration constant | The action performed has a high risk of losing data, such as reformatting a hard disk. |
| Low | Enumeration constant | The action performed has a low risk of losing data. |
| Medium | Enumeration constant | The action performed has a medium risk of losing data. |
| None | Enumeration constant | Do not confirm any actions (suppress all requests for confirmation). |

Windows PowerShell: This type is System.Management.Automation.ConfirmImpact.

#### File-Attributes type

This implementation-defined type has the following accessible members, which can be combined:

| **Member** | **Member Kind** | **Purpose** |
| --- | --- | --- |
| Archive | Enumeration constant | The file's archive status. Applications use this attribute to mark files for backup or removal. |
| Compressed | Enumeration constant | The file is compressed. |
| Device |  | Reserved for future use. |
| Directory | Enumeration constant | The file is a directory. |
| Encrypted | Enumeration constant | The file or directory is encrypted. For a file, this means that all data in the file is encrypted. For a directory, this means that encryption is the default for newly created files and directories. |
| Hidden | Enumeration constant | The file is hidden, and thus is not included in an ordinary directory listing. |
| Normal | Enumeration constant | The file is normal and has no other attributes set. This attribute is valid only if used alone. |
| NotContentIndexed | Enumeration constant | The file will not be indexed by the operating system's content indexing service. |
| Offline | Enumeration constant | The file is offline. The data of the file is not immediately available. |
| ReadOnly | Enumeration constant | The file is read-only. |
| ReparsePoint | Enumeration constant | The file contains a reparse point, which is a block of user-defined data associated with a file or a directory. |
| SparseFile | Enumeration constant | The file is a sparse file. Sparse files are typically large files whose data are mostly zeros. |
| System | Enumeration constant | The file is a system file. The file is part of the operating system or is used exclusively by the operating system. |
| Temporary | Enumeration constant | The file is temporary. File systems attempt to keep all of the data in memory for quicker access rather than flushing the data back to mass storage. A temporary file should be deleted by the application as soon as it is no longer needed. |
| ReparsePoint | Enumeration constant | The file contains a reparse point, which is a block of user-defined data associated with a file or a directory. |

Windows PowerShell: This type is System.IO.FileAttributes with attribute FlagsAttribute.

#### Regular-Expression-Option type

This implementation-defined type has the following accessible members, which can be combined:

| **Member** | **Member Kind** | **Purpose** |
| --- | --- | --- |
| IgnoreCase | Enumeration constant | Specifies that the matching is case-insensitive. |
| None | Enumeration constant | Specifies that no options are set. |

An implementation may provide other values.

Windows PowerShell: This type is System.Text.RegularExpressions.RegexOptions with attribute FlagsAttribute.

Windows PowerShell: The following extra values are defined: Compiled, CultureInvariant, ECMAScript, ExplicitCapture, IgnorePatternWhitespace, Multiline, RightToLeft, Singleline.

## Reference types

### Strings

A string value has type string and is an immutable sequence of zero or more characters of type char each containing a UTF-16-encoded 16-bit Unicode code point.

Type string has the following accessible members:

| **Member** | **Member Kind** | **Type** | **Purpose** |
| --- | --- | --- | --- |
| Length | Instance Property | int (read-only) | Gets the number of characters in the string |
| ToLower | Instance Method | string | Creates a new string that contains the lowercase equivalent |
| ToUpper | Instance Method | string | Creates a new string that contains the uppercase equivalent |

Windows PowerShell: string maps to System.String.

### Arrays

All array types are derived from the type Array. This type has the following accessible members:

| **Member** | **Member Kind** | **Type** | **Purpose** |
| --- | --- | --- | --- |
| Length | Instance Property (read-only) | int | Number of elements in the array |
| Rank | Instance Property (read-only) | int | Number of dimensions in the array |
| Copy | Static Method | void/see Purpose column | Copies a range of elements from one array to another. There are four versions, where source is the source array, destination is the destination array, count is the number of elements to copy, and sourceIndex and destinationIndex are the starting locations in their respective arrays:  Copy(source, destination, int count) Copy(source, destination, long count) Copy(source, sourceIndex, destination, destinationIndex, int count) Copy(source, sourceIndex, destination, destinationIndex, long count) |
| GetLength | Instance Method (read-only) | int/none | Number of elements in a given dimension  GetLength(int dimension) |

For more details on arrays, see §9.

Windows PowerShell: Array maps to System.Array.

### Hashtables

Type Hashtable has the following accessible members:

| **Member** | **Member Kind** | **Type** | **Purpose** |
| --- | --- | --- | --- |
| Count | Instance Property | int | Gets the number of key/value pairs in the Hashtable |
| Keys | Instance Property | Implementation-defined | Gets a collection of all the keys |
| Values | Instance Property | Implementation-defined | Gets a collection of all the values |
| Remove | Instance Method | void/none | Removes the designated key/value |

For more details on Hashtables, see §10.

Windows PowerShell: Hashtable maps to System.Collections.Hashtable. Hashtable elements are stored in an object of type DictionaryEntry, and the collections returned by Keys and Values have type ICollection.

### The xml type

Type xml implements the W3C Document Object Model (DOM) Level 1 Core and the Core DOM Level 2. The DOM is an in-memory (cache) tree representation of an XML document and enables the navigation and editing of this document. This type supports the subscript operator [] (§7.1.4.4).

Windows PowerShell: xml maps to System.Xml.XmlDocument.

### The regex type

Type regex provides machinery for supporting regular expression processing. It is used to constrain the type of a parameter (§5.3) whose corresponding argument might contain a regular expression.

Windows PowerShell: regex maps to System.Text.RegularExpressions.Regex.

### The ref type

Ordinarily, arguments are passed to commands by value. In the case of an argument having some value type a copy of the value is passed. In the case of an argument having some reference type a copy of the reference is passed.

Type ref provides machinery to allow arguments to be passed to commands by reference, so the commands can modify the argument's value. Type ref has the following accessible members:

| **Member** | **Member Kind** | **Type** | **Purpose** |
| --- | --- | --- | --- |
| Value | Instance property (read-write) | The type of the value being referenced. | Gets/sets the value being referenced. |

Consider the following function definition and call:

function Doubler  
{

param ([ref]$x) # parameter received by reference  
  
 $x.Value \*= 2.0 # note that 2.0 has type double  
}

$number = 8 # designates a value of type int, value 8  
Doubler([ref]$number) # argument received by reference  
$number # designates a value of type double, value 8.0

Consider the case in which $number is type-constrained:

[int]$number = 8 # designates a value of type int, value 8  
Doubler([ref]$number) # argument received by reference  
$number # designates a value of type int, value 8

As shown, both the argument and its corresponding parameter must be declared ref.

Windows PowerShell: regex maps to System.Management.Automation.PSReference.

### The scriptblock type

Type scriptblock represents a precompiled block of script text (§7.1.8) that can be used as a single unit. It has the following accessible members:

| **Member** | **Member Kind** | **Type** | **Purpose** |
| --- | --- | --- | --- |
| Attributes | Instance property (read-only) | Collection of attributes | Gets the attributes of the script block. |
| File | Instance property (read-only) | string | Gets the name of the file in which the script block is defined. |
| Module | Instance property (read-only) | implementation defined (§4.5.12) | Gets information about the module in which the script block is defined. |
| GetNewClosure | Instance method | scriptblock /none | Retrieves a script block that is bound to a module. Any local variables that are in the context of the caller will be copied into the module. |
| Invoke | Instance method | Collection of object/object[] | Invokes the script block with the specified arguments and returns the results. |
| InvokeReturnAsIs | Instance method | object/object[] | Invokes the script block with the specified arguments and returns any objects generated. |
| Create | Static method | scriptblock /string | Creates a new scriptblock object that contains the specified script. |

Windows PowerShell: scriptblock maps to System.Management.Automation.ScriptBlock.

Windows PowerShell: Invoke returns a collection of PsObject.

### The math type

Type math provides access to some constants and methods useful in mathematical computations. It has the following accessible members:

| **Member** | **Member Kind** | **Type** | **Purpose** |
| --- | --- | --- | --- |
| E | Static property (read-only) | double | Natural logarithmic base |
| PI | Static property (read-only) | double | Ratio of the circumference of a circle to its diameter |
| Abs | Static method | numeric/numeric | Absolute value (the return type is the same as the type of the argument passed in) |
| Acos | Static method | double / double | Angle whose cosine is the specified number |
| Asin | Static method | double / double | Angle whose sine is the specified number |
| Atan | Static method | double / double | Angle whose tangent is the specified number |
| Atan2 | Static method | double / double y, double x | Angle whose tangent is the quotient of two specified numbers x and y |
| Ceiling | Static method | decimal / decimal  double / double | smallest integer greater than or equal to the specified number |
| Cos | Static method | double / double | Cosine of the specified angle |
| Cosh | Static method | double / double | Hyperbolic cosine of the specified angle |
| Exp | Static method | double / double | e raised to the specified power |
| Floor | Static method | decimal / decimal  double / double | Largest integer less than or equal to the specified number |
| Log | Static method | double / double number  double / double number, double base | Logarithm of number using base e or base base |
| Log10 | Static method | double / double | Base-10 logarithm of a specified number |
| Max | Static method | numeric/numeric | Larger of two specified numbers (the return type is the same as the type of the arguments passed in) |
| Min | Static method | numeric/numeric, numeric | Smaller of two specified numbers (the return type is the same as the type of the arguments passed in) |
| Pow | Static method | double / double x, double y | A specified number x raised to the specified power y |
| Sin | Static method | double / double | Sine of the specified angle |
| Sinh | Static method | double / double | Hyperbolic sine of the specified angle |
| Sqrt | Static method | double / double | Square root of a specified number |
| Tan | Static method | double / double | Tangent of the specified angle |
| Tanh | Static method | double / double | Hyperbolic tangent of the specified angle |

Windows PowerShell: Math maps to System.Math.

## Generic types

A number of programming languages and environments provide types that can be specialized. Many of these types are referred to as container types, as instances of them are able to contain objects of some other type. Consider a type called Stack that can represent a stack of values, which can be pushed on and popped off. Typically, the user of a stack wants to store only one kind of object on that stack. However, if the language or environment does not support type specialization, multiple distinct variants of the type Stack must be implemented even though they all perform the same task, just with different type elements.

Type specialization allows a generic type to be implemented such that it can be constrained to handling some subset of types when it is used. For example,

* A generic stack type that is specialized to hold strings might be written as Stack[string].
* A generic dictionary type that is specialized to hold int keys with associated string values might be written as Dictionary[int,string].
* A stack of stack of strings might be written as Stack[Stack[string]].

Although PowerShell does not define any built-in generic types, it can use such types if they are provided by the host environment. See the syntax in §7.1.10.

Windows PowerShell: The complete name for the type Stack[string] suggested above is System.Collections.Generic.Stack[int]. The complete name for the type Dictionary[int,string] suggested above is System.Collections.Generic.Dictionary[int,string].

## Anonymous types

In some circumstances, an implementation of PowerShell creates objects of some type, and those objects have members accessible to script. However, the actual name of those types need not be specified, so long as the accessible members are specified sufficiently for them to be used. That is, scripts can save objects of those types and access their members without actually knowing those types' names. The following subsections specify these types.

### Provider description type

This type encapsulates the state of a provider. It has the following accessible members:

| **Member** | **Member Kind** | **Type** | **Purpose** |
| --- | --- | --- | --- |
| Drives | Instance property (read-only) | Implementation defined (§4.5.2) | A collection of drive description objects |
| Name | Instance property (read-only) | string | The name of the provider |

Windows PowerShell: This type is System.Management.Automation.ProviderInfo.

### Drive description type

This type encapsulates the state of a drive. It has the following accessible members:

| **Member** | **Member Kind** | **Type** | **Purpose** |
| --- | --- | --- | --- |
| CurrentLocation | Instance property (read-write) | string | The current working location (§3.1.4) of the drive |
| Description | Instance property (read-write) | string | The description of the drive |
| Name | Instance property (read-only) | string | The name of the drive |
| Root | Instance property (read-only) | string | The name of the drive |

Windows PowerShell: This type is System.Management.Automation.PSDriveInfo.

### Variable description type

This type encapsulates the state of a variable. It has the following accessible members:

| **Member** | **Member Kind** | **Type** | **Purpose** |
| --- | --- | --- | --- |
| Attributes | Instance property (read-only) | Implementation defined | A collection of attributes |
| Description | Instance property (read-write) | string | The description assigned to the variable via the New-Variable (§13.37) or Set-Variable (§13.50) cmdlets. |
| Module | Instance property (read-only) | Implementation defined (§4.5.12) | The module from which this variable was exported |
| ModuleName | Instance property (read-only) | string | The module in which this variable was defined |
| Name | Instance property (read-only) | string | The name assigned to the variable when it was created in the PowerShell language or via the New-Variable (§13.37) and Set-Variable (§13.50) cmdlets. |
| Options | Instance property (read-write) | string | The options assigned to the variable via the New-Variable (§13.37) or Set-Variable (§13.50) cmdlets. |
| Value | Instance property (read-write) | object | The value assigned to the variable when it was assigned in the PowerShell language or via the New-Variable (§13.37) and Set-Variable (§13.50) cmdlets. |

Windows PowerShell: This type is System.Management.Automation.PSVariable.

Windows PowerShell: The type of the attribute collection is System.Management.Automation.PSVariableAttributeCollection.

### Alias description type

This type encapsulates the state of an alias. It has the following accessible members:

| **Member** | **Member Kind** | **Type** | **Purpose** |
| --- | --- | --- | --- |
| CommandType | Instance property (read-only) | Implementation defined | Should compare equal with "Alias". |
| Definition | Instance property (read-only) | string | The command or alias to which the alias was assigned via the New-Alias (§13.33) or Set-Alias (§13.46) cmdlets. |
| Description | Instance property (read-write) | string | The description assigned to the alias via the New-Alias (§13.33) or Set-Alias (§13.46) cmdlets. |
| Module | Instance property (read-only) | Implementation defined (§4.5.12) | The module from which this alias was exported |
| ModuleName | Instance property (read-only) | string | The module in which this alias was defined |
| Name | Instance property (read-only) | string | The name assigned to the alias when it was created via the New-Alias (§13.33) or Set-Alias (§13.46) cmdlets. |
| Options | Instance property (read-write) | string | The options assigned to the alias via the New-Alias (§13.33) or Set-Alias (§13.46) cmdlets. |
| OutputType | Instance property (read-only) | Implementation defined collection | Specifies the types of the values output by the command to which the alias refers. |
| Parameters | Instance property (read-only) | Implementation defined collection | The parameters of the command. |
| ParameterSets | Instance property (read-only) | Implementation defined collection | Information about the parameter sets associated with the command. |
| ReferencedCommand | Instance property (read-only) | Implementation defined | Information about the command that is immediately referenced by this alias. |
| ResolvedCommand | Instance property (read-only) | Implementation defined | Information about the command to which the alias eventually resolves. |

Windows PowerShell: This type is System.Management.Automation.AliasInfo.

### Working location description type

This type encapsulates the state of a working location. It has the following accessible members:

| **Member** | **Member Kind** | **Type** | **Purpose** |
| --- | --- | --- | --- |
| Drive | Instance property (read-only) | Implementation defined (§4.5.2) | A drive description object |
| Path | Instance property (read-only) | string | The working location |
| Provider | Instance property (read-only) | Implementation defined (§4.5.1) | The provider |
| ProviderPath | Instance property (read-only) | string | The current path of the provider |

A stack of working locations is a collection of working location objects, as described above.

Windows PowerShell: A current working location is represented by an object of type System.Management.Automation.PathInfo.

Windows PowerShell: A stack of working locations is represented by an object of type System.Management.Automation.PathInfoStack, which is a collection of PathInfo objects.

### Environment variable description type

This type encapsulates the state of an environment variable. It has the following accessible members:

| **Member** | **Member Kind** | **Type** | **Purpose** |
| --- | --- | --- | --- |
| Name | Instance property (read-write) | string | The name of the environment variable |
| Value | Instance property (read-write) | string | The value of the environment variable |

Windows PowerShell: This type is System.Collections.DictionaryEntry. The name of the variable is the dictionary key. The value of the environment variable is the dictionary value. Name is an AliasProperty that equates to Key.

### Application description type

This type encapsulates the state of an application. It has the following accessible members:

| **Member** | **Member Kind** | **Type** | **Purpose** |
| --- | --- | --- | --- |
| CommandType | Instance property (read-only) | Implementation defined | Should compare equal with "Application". |
| Definition | Instance property (read-only) | string | A description of the application. |
| Extension | Instance property (read-write) | string | The extension of the application file. |
| Module | Instance property (read-only) | Implementation defined (§4.5.12) | The module that defines this command. |
| ModuleName | Instance property (read-only) | string | The name of the module that defines the command. |
| Name | Instance property (read-only) | string | The name of the command. |
| OutputType | Instance property (read-only) | Implementation defined collection | Specifies the types of the values output by the command. |
| Parameters | Instance property (read-only) | Implementation defined collection | The parameters of the command. |
| ParameterSets | Instance property (read-only) | Implementation defined collection | Information about the parameter sets associated with the command. |
| Path | Instance property (read-only) | string | Gets the path of the application file. |

Windows PowerShell: This type is System.Management.Automation.ApplicationInfo.

### Cmdlet description type

This type encapsulates the state of a cmdlet. It has the following accessible members:

| **Member** | **Member Kind** | **Type** | **Purpose** |
| --- | --- | --- | --- |
| CommandType | Instance property (read-only) | Implementation defined | Should compare equal with "Cmdlet". |
| DefaultParameterSet | Instance property (read-only) | Implementation defined | The default parameter set that is used if PowerShell cannot determine which parameter set to use based on the supplied arguments. |
| Definition | Instance property (read-only) | string | A description of the cmdlet. |
| HelpFile | Instance property (read-write) | string | The path to the Help file for the cmdlet. |
| ImplementingType | Instance property (read-write) | Implementation defined | The type that implements the cmdlet. |
| Module | Instance property (read-only) | Implementation defined (§4.5.12) | The module that defines this cmdlet. |
| ModuleName | Instance property (read-only) | string | The name of the module that defines the cmdlet. |
| Name | Instance property (read-only) | string | The name of the cmdlet. |
| Noun | Instance property (read-only) | string | The noun name of the cmdlet. |
| OutputType | Instance property (read-only) | Implementation defined collection | Specifies the types of the values output by the cmdlet. |
| Parameters | Instance property (read-only) | Implementation defined collection | The parameters of the cmdlet. |
| ParameterSets | Instance property (read-only) | Implementation defined collection | Information about the parameter sets associated with the cmdlet. |
| Verb | Instance property (read-only) | string | The verb name of the cmdlet. |
| PSSnapIn | Instance property (read-only) | Implementation defined | Windows PowerShell: Information about the Windows Powershell snap-in that is used to register the cmdlet. |

Windows PowerShell: This type is System.Management.Automation.CmdletInfo.

### External script description type

This type encapsulates the state of an external script (one that is directly executable by PowerShell, but is not built in). It has the following accessible members:

| **Member** | **Member Kind** | **Type** | **Purpose** |
| --- | --- | --- | --- |
| CommandType | Instance property (read-only) | Implementation defined | Should compare equal with "ExternalScript". |
| Definition | Instance property (read-only) | string | A definition of the script. |
| Module | Instance property (read-only) | Implementation defined (§4.5.12) | The module that defines this script. |
| ModuleName | Instance property (read-only) | string | The name of the module that defines the script. |
| Name | Instance property (read-only) | string | The name of the script. |
| OriginalEncoding | Instance property (read-only) | Implementation defined | The original encoding used to convert the characters of the script to bytes. |
| OutputType | Instance property (read-only) | Implementation defined collection | Specifies the types of the values output by the script. |
| Parameters | Instance property (read-only) | Implementation defined collection | The parameters of the script. |
| ParameterSets | Instance property (read-only) | Implementation defined collection | Information about the parameter sets associated with the script. |
| Path | Instance property (read-only) | string | The path to the script file. |
| ScriptBlock | Instance property (read-only) | scriptblock | The external script. |
| ScriptContents | Instance property (read-only) | string | The original contents of the script. |

Windows PowerShell: This type is System.Management.Automation.ExternalScriptInfo.

### Function description type

This type encapsulates the state of a function. It has the following accessible members:

| **Member** | **Member Kind** | **Type** | **Purpose** |
| --- | --- | --- | --- |
| CmdletBinding | Instance property (read-only) | bool | Indicates whether the function uses the same parameter binding that compiled cmdlets use (see §12.3.5). |
| CommandType | Instance property (read-only) | Implementation defined | Can be compared for equality with "Function" or "Filter" to see which of those this object represents. |
| DefaultParameterSet | Instance property (read-only) | string | Specifies the parameter set to use if that cannot be determined from the arguments (see §12.3.5). |
| Definition | Instance property (read-only) | string | A string version of ScriptBlock |
| Description | Instance property (read-write) | string | The description of the function. |
| Module | Instance property (read-only) | Implementation defined (§4.5.12) | The module from which this function was exported |
| ModuleName | Instance property (read-only) | string | The module in which this function was defined |
| Name | Instance property (read-only) | string | The name of the function |
| Options | Instance property (read-write) | Implementation defined | The scope options for the function (§3.5.4). |
| OutputType | Instance property (read-only) | Implementation defined collection | Specifies the types of the values output, in order (see §12.3.6). |
| Parameters | Instance property (read-only) | Implementation defined collection | Specifies the parameter names, in order. If the function acts like a cmdlet (see CmdletBinding above) the common parameters (§13.56) are included at the end of the collection. |
| ParameterSets | Instance property (read-only) | Implementation defined collection | Information about the parameter sets associated with the command. For each parameter, the result shows the parameter name and type, and indicates if the parameter is mandatory, by position or a switch parameter. If the function acts like a cmdlet (see CmdletBinding above) the common parameters (§13.56) are included at the end of the collection. |
| ScriptBlock | Instance property (read-only) | scriptblock (§4.3.6) | The body of the function |

Windows PowerShell: This type is System.Management.Automation.FunctionInfo.

CommandType has type System.Management.Automation.CommandTypes.

Options has type System.Management.Automation.ScopedItemOptions.

OutputType has type System.Collections.ObjectModel.ReadOnlyCollection`1[[System.Management.Automation.PSTypeName, System.Management.Automation]].

Parameters has type System.Collections.Generic.Dictionary`2[[System.String, mscorlib],[System.Management.Automation.ParameterMetadata,System.Management.Automation]].

ParameterSets has type System.Collections.ObjectModel.ReadOnlyCollection`1[[System.Management.Automation.CommandParameterSetInfo, System.Management.Automation]].

Visibility has type System.Management.Automation.SessionStateEntryVisibility.

Windows PowerShell also has a property called Visibility.

### Filter description type

This type encapsulates the state of a filter. It has the same set of accessible members as the function description type (§4.5.10).

Windows PowerShell: This type is System.Management.Automation.FilterInfo. It has the same set of properties as System.Management.Automation.FunctionInfo (§4.5.11).

### Module description type

This type encapsulates the state of a module. It has the following accessible members:

| **Member** | **Member Kind** | **Type** | **Purpose** |
| --- | --- | --- | --- |
| Description | Instance property (read-write) | string | The description of the module (set by the manifest) |
| ModuleType | Instance property (read-only) | Implementation defined | The type of the module (Manifest, Script, or Binary) |
| Name | Instance property (read-only) | string | The name of the module |
| Path | Instance property (read-only) | string | The module's path |

Windows PowerShell: This type is System.Management.Automation.PSModuleInfo.

Windows PowerShell: The type of ModuleType is System.Management.Automation.ModuleType.

### Custom object description type

This type encapsulates the state of a custom object. It has no accessible members.

Windows PowerShell: This type is System.Management.Automation.PSCustomObject. The cmdlets Import-Module and New-Object can generate an object of this type.

### Command description type

The automatic variable $PsCmdlet is an object that represents the cmdlet or function being executed. The type of this object is implementation defined; it has the following accessible members:

| **Member** | **Member Kind** | **Type** | **Purpose** |
| --- | --- | --- | --- |
| ParameterSetName | Instance property (read-only) | string | Name of the current parameter set (see ParameterSetName) |
| ShouldContinue | Instance method | Overloaded  /bool | Requests confirmation of an operation from the user. |
| ShouldProcess | Instance method | Overloaded  /bool | Requests confirmation from the user before an operation is performed. |

Windows PowerShell: This type is System.Management.Automation.PSScriptCmdlet.

### Error record description type

The automatic variable $Error contains a collection of error records that represent recent errors (§3.12). Although the type of this collection is unspecified, it does support subscripting to get access to individual error records.

Windows PowerShell: The collection type is System.Collections.ArrayList.

Windows PowerShell: The type of an individual error record in the collection is System.Management.Automation.ErrorRecord. This type has the following public properties:

CategoryInfo – Gets information about the category of the error.

ErrorDetails – Gets and sets more detailed error information, such as a replacement error message.

Exception – Gets the exception that is associated with this error record.

FullyQualifiedErrorId – Gets the fully qualified error identifier for this error record.

InvocationInfo – Gets information about the command that was invoked when the error occurred.

PipelineIterationInfo – Gets the status of the pipeline when this error record was created

TargetObject – Gets the object that was being processed when the error occurred.

### Enumerator description type

A number of variables are enumerators for collections (§4). The automatic variable $foreach is the enumerator created for any foreach statement. The automatic variable $input is the enumerator for a collection delivered to a function from the pipeline. The automatic variable $switch is the enumerator created for any switch statement.

The type of an enumerator is implementation defined; it has the following accessible members:

| **Member** | **Member Kind** | **Type** | **Purpose** |
| --- | --- | --- | --- |
| Current | Instance property (read-only) | object | Gets the current element in the collection. If the enumerator is not currently positioned at an element of the collection, the behavior is implementation defined. |
| MoveNext | Instance method | None/bool | Advances the enumerator to the next element of the collection. Returns $true if the enumerator was successfully advanced to the next element; $false if the enumerator has passed the end of the collection. |

Windows PowerShell: These members are defined in the interface System.IEnumerator, which is implemented by the types identified below.

Windows PowerShell: If the enumerator is not currently positioned at an element of the collection, an exception of type InvalidOperationException is raised.

Windows PowerShell: For $foreach, this type is System.Array+SZArrayEnumerator.

Windows PowerShell: For $input, this type is System.Collections.ArrayList+ArrayListEnumeratorSimple.

Windows PowerShell: For $switch, this type is System.Array+SZArrayEnumerator.

### Directory description type

The cmdlet New-Item (§13.34) can create items of various kinds including FileSystem directories. The type of a directory description object is implementation defined; it has the following accessible members:

| **Member** | **Member Kind** | **Type** | **Purpose** |
| --- | --- | --- | --- |
| Attributes | Instance property (read-write) | Implementation defined (§4.2.6.3) | Gets or sets one or more of the attributes of the directory object. |
| CreationTime | Instance property (read-write) | Implementation defined (§4.5.19) | Gets and sets the creation time of the directory object. |
| Extension | Instance property (read- only) | string | Gets the extension part of the directory name. |
| FullName | Instance property (read-only) | string | Gets the full path of the directory. |
| LastWriteTime | Instance property (read-write) | Implementation defined (§4.5.19) | Gets and sets the time when the directory was last written to. |
| Name | Instance property (read- only) | string | Gets the name of the directory. |

Windows PowerShell: This type is System.IO.DirectoryInfo. The type of the Attributes property is System.IO.FileAttributes.

### File description type

The cmdlet New-Item (§13.34) can create items of various kinds including FileSystem files. The type of a file description object is implementation defined; it has the following accessible members:

| **Member** | **Member Kind** | **Type** | **Purpose** |
| --- | --- | --- | --- |
| Attributes | Instance property (read-write) | Implementation defined (§4.2.6.3) | Gets or sets one or more of the attributes of the file object. |
| BaseName | Instance property (read- only) | string | Gets the name of the file excluding the extension. |
| CreationTime | Instance property (read-write) | Implementation defined (§4.5.19) | Gets and sets the creation time of the file object. |
| Extension | Instance property (read- only) | string | Gets the extension part of the file name. |
| FullName | Instance property (read-only) | string | Gets the full path of the file. |
| LastWriteTime | Instance property (read-write) | Implementation defined (§4.5.19) | Gets and sets the time when the file was last written to. |
| Length | Instance property (read- only) | long | Gets the size of the file, in bytes. |
| Name | Instance property (read- only) | string | Gets the name of the file. |
| VersionInfo | Instance property (read- only) | Implementation defined | Windows PowerShell: This ScriptProperty returns a System.Diagnostics.FileVersionInfo for the file. |

Windows PowerShell: This type is System.IO.FileInfo.

### Date-Time description type

The type of a date-time description object is implementation defined; it has the following accessible members:

| **Member** | **Member Kind** | **Type** | **Purpose** |
| --- | --- | --- | --- |
| Day | Instance property (read-only) | int | Gets the day component of the month represented by this instance. |
| Hour | Instance property (read-only) | int | Gets the hour component of the date represented by this instance. |
| Minute | Instance property (read-only) | int | Gets the minute component of the date represented by this instance. |
| Month | Instance property (read-only) | int | Gets the month component of the date represented by this instance. |
| Second | Instance property (read-only) | int | Gets the seconds component of the date represented by this instance. |
| Year | Instance property (read-only) | int | Gets the year component of the date represented by this instance. |

An object of this type can be created by cmdlet Get-Date (§13.18).

Windows PowerShell: This type is System.DateTime.

### Group-Info description type

The type of a group-info description object is implementation defined; it has the following accessible members:

| Member | Member Kind | Type | Purpose |
| --- | --- | --- | --- |
| Count | Instance property (read-only) | int | Gets the number of elements in the group. |
| Group | Instance property (read-only) | Implementation-defined collection | Gets the elements of the group. |
| Name | Instance property (read-only) | string | Gets the name of the group. |
| Values | Instance property (read-only) | Implementation-defined collection | Gets the values of the elements of the group. |

An object of this type can be created by cmdlet Group-Object (§13.27).

Windows PowerShell: This type is Microsoft.PowerShell.Commands.GroupInfo.

### Generic-Measure-Info description type

The type of a generic-measure-info description object is implementation defined; it has the following accessible members:

| **Member** | **Member Kind** | **Type** | **Purpose** |
| --- | --- | --- | --- |
| Average | Instance property (read-only) | double | Gets the average of the values of the properties that are measured. |
| Count | Instance property (read-only) | int | Gets the number of objects with the specified properties. |
| Maximum | Instance property (read-only) | double | Gets the maximum value of the specified properties. |
| Minimum | Instance property (read-only) | double | Gets the minimum value of the specified properties. |
| Property | Instance property (read-only) | string | Gets the property to be measured. |
| Sum | Instance property (read-only) | double | Gets the sum of the values of the specified properties. |

An object of this type can be created by cmdlet Measure-Object (§13.31).

Windows PowerShell: This type is Microsoft.PowerShell.Commands.GenericMeasureInfo.

### Text-Measure-Info description type

The type of a text-info description object is implementation defined; it has the following accessible members:

| **Member** | **Member Kind** | **Type** | **Purpose** |
| --- | --- | --- | --- |
| Characters | Instance property (read-only) | int | Gets the number of characters in the target object. |
| Lines | Instance property (read-only) | int | Gets the number of lines in the target object. |
| Property | Instance property (read-only) | string | Gets the property to be measured. |
| Words | Instance property (read-only) | int | Gets the number of words in the target object. |

An object of this type can be created by cmdlet Measure-Object (§13.31).

Windows PowerShell: This type is Microsoft.PowerShell.Commands.TextMeasureInfo.

### Credential type

A credential object can then be used in various security operations. The type of a credential object is implementation defined; it has the following accessible members:

| Member | Member Kind | Type | Purpose |
| --- | --- | --- | --- |
| Password | Instance property (read-only) | Implementation defined | Gets the password. |
| UserName | Instance property (read-only) | string | Gets the username. |

An object of this type can be created by cmdlet Get-Credential (§13.17).

Windows PowerShell: This type is System.Management.Automation.PSCredential.

### Method designator type

The type of a method designator is implementation defined; it has the following accessible members:

| **Member** | **Member Kind** | **Type** | **Purpose** |
| --- | --- | --- | --- |
| Invoke | Instance method | object/variable number and type | Takes a variable number of arguments, and indirectly calls the method referred to by the parent method designator, passing in the arguments. |

An object of this type can be created by an invocation-expression (§7.1.3).

Windows PowerShell: This type is System.Management.Automation.PSMethod.

### Member definition type

This type encapsulates the definition of a member. It has the following accessible members:

| **Member** | **Member Kind** | **Type** | **Purpose** |
| --- | --- | --- | --- |
| Definition | Instance property (read-only) | string | Gets the definition of the member. |
| MemberType | Instance property (read-only) | Implementation defined | Gets the PowerShell type of the member. |
| Name | Instance property (read-only) | string | Gets the name of the member. |
| TypeName | Instance property (read-only) | string | Gets the type name of the member. |

Windows PowerShell: This type is Microsoft.PowerShell.Commands.MemberDefinition.

## Type extension and adaptation

A PowerShell implementation includes a family of core types (which are documented in this chapter) that each contain their own set of base members. Those members can be methods or properties, and they can be instance or static members. For example, the base members of the type string (§4.3.1) are the instance property Length and the instance methods ToLower and ToUpper.

When an object is created, it contains all the instance properties of that object's type, and the instance methods of that type can be called on that object. An object may be customized via the addition of instance members at runtime. The result is called a custom object. Any members added to an instance exist only for the life of that instance; other instances of the same core type are unaffected.

The base member set of a type can be augmented by the addition of the following kinds of members:

* adapted members, via the Extended Type System (ETS), most details of which are unspecified.
* extended members, via the cmdlet Add-Member (§13.2).

Windows PowerShell: Extended members can also be added via types.ps1xml files.

Adapted and extended members are collectively called synthetic members.

The ETS adds the following members to all PowerShell objects: psbase, psadapted, psextended, and pstypenames. (See the Force and View parameters in the cmdlet Get-Member (§13.22) for more information on these members.)

An instance member may hide an extended and/or adapted member of the same name, and an extended member may hide an adapted member. In such cases, the member sets psadapted and psextended can be used to access those hidden members.

Windows PowerShell: If a types.ps1xml specifies a member called Supports, obj.psextended provides access to just that member and not to a member added via Add-Member.

There are three ways create a custom object having a new member M:

1. $x = New-Object PsObject –Property @{M = 123}  
   This approach can be used to add one or more NoteProperty members.
2. $x = New-Module –AsCustomObject {$M = 123 ; Export-ModuleMember –Variable M}  
   This approach can be used to add NoteProperty or ScriptMethod members.
3. $x = New-Object PsObject  
   Add-Member –InputObject $x –Name M –MemberType NoteProperty –Value 123  
     
   This approach can be used to add any kind of member.

Windows PowerShell: PsObject is the base type of all PowerShell types.

# Variables

A variable represents a storage location for a value, and that value has a type. Traditional procedural programming languages are statically typed; that is, the runtime type of a variable is that with which it was declared at compile time. Object-oriented languages add the idea of inheritance, which allows the runtime type of a variable to be that with which it was declared at compile time or some type derived from that type. Being a dynamically typed language, PowerShell's variables do not have types, per se. In fact, variables are not defined; they simply come into being when they are first assigned a value. And while a variable may be constrained (§5.3) to holding a value of a given type, type information in an assignment cannot always be verified statically.

At different times, a variable may be associated with values of different types either through assignment (§7.11) or the use of the ++ and ‑‑ operators (§7.1.5, 7.2.6). When the value associated with a variable is changed, that value's type may change. For example,

$i = "abc" # $i holds a value of type string  
$i = 2147483647 # $i holds a value of type int  
++$i # $i now holds a value of type double because  
 # 2147483648 is too big to fit in type int

Any use of a variable that has not been created results in the value $null. [Note: To see if a variable has been defined, use the Test-Path cmdlet (§13.54). end note]

## Writable location

A writable location is an expression that designates a resource to which a command has both read and write access. A writable location may be a variable (§5), an array element (§9), an associated value in a Hashtable accessed via a subscript (§10), a property (§7.1.2), or storage managed by a provider (§3.1).

## Variable categories

PowerShell defines the following categories of variables: static variables, instance variables, array elements, Hashtable key/value pairs, parameters, ordinary variables, and variables on provider drives. The subsections that follow describe each of these categories.

In the following example

function F ($p1, $p2)  
{  
 $radius = 2.45  
 $circumference = 2 \* ([Math]::PI) \* $radius

$date = Get-Date -Date "2010-2-1 10:12:14 pm"  
 $month = $date.Month

$values = 10,55,93, 102  
 $value = $values[2]

$h1 = @{ FirstName = "James"; LastName = "Anderson" }  
 $h1.FirstName = "Smith"

$Alias:A = "Help"  
 $Env:MyPath = "e:\Temp"  
 ${E:output.txt} = 123  
 $function:F = { "Hello there" }  
 $Variable:v = 10  
}

* [Math::PI] is a static variable
* $date.Month is an instance variable
* $values[2] is an array element
* $h1.FirstName is a Hashtable key whose corresponding value is $h1['FirstName']
* $p1 and $p2 are parameters
* $radius, $circumference, $date, $month, $values, $value, and $h1 are ordinary variables
* $Alias:A, $Env:MyPath, ${E:output.txt}, and $function:F are variables on the corresponding provider drives.
* $Variable:v is actually an ordinary variable written with its fully qualified provider drive.

### Static variables

A data member of an object that belongs to the object's type rather than to that particular instance of the type is called a static variable. See §4.2.3, §4.2.4.1, and §4.3.8 for some examples.

PowerShell provides no way to create new types that contain static variables; however, objects of such types may be provided by the host environment.

Memory for creating and deleting objects containing static variables is managed by the host environment and the garbage collection system.

See §7.1.2 for information about accessing a static variable.

Windows PowerShell: A static data member can be a field or a property.

### Instance variables

A data member of an object that belongs to a particular instance of the object's type rather than to the type itself is called an instance variable. See §4.3.1, §4.3.2, and §4.3.3 for some examples.

A PowerShell host environment might provide a way to create new types that contain instance variables or to add new instance variables to existing types.

Memory for creating and deleting objects containing static variables is managed by the host environment and the garbage collection system.

See §7.1.2 for information about accessing an instance variable.

Windows PowerShell: An instance data member can be a field or a property.

### Array elements

An array can be created via a unary comma operator (§7.2.1), sub-expression (§7.1.6), array-expression (§7.1.7), binary comma operator (§7.3), range operator (§7.4), or New-Object cmdlet (§13.36).

Memory for creating and deleting arrays is managed by the host environment and the garbage collection system.

Arrays and array elements are discussed in §9.

### Hashtable key/value pairs

A Hashtable is created via a hash literal (§2.3.5.6) or the New-Object cmdlet (§13.36). A new key/value pair can be added via the [] operator (§7.1.4.3).

Memory for creating and deleting Hashtables is managed by the host environment and the garbage collection system.

Hashtables are discussed in §10.

### Parameters

A parameter is created when its parent command is invoked, and it is initialized with the value of the argument provided in the invocation or by the host environment. A parameter ceases to exist when its parent command terminates.

Parameters are discussed in §8.10.

### Ordinary variables

An ordinary variable is defined by an assignment-expression (§7.11) or a foreach-statement (§8.4.4). Some ordinary variables are predefined by the host environment while others are transient, coming and going as needed at runtime.

The lifetime of an ordinary variable is that part of program execution during which storage is guaranteed to be reserved for it. This lifetime begins at entry into the scope with which it is associated, and ends no sooner than the end of the execution of that scope. If the parent scope is entered recursively or iteratively, a new instance of the local variable is created each time.

The storage referred to by an ordinary variable is reclaimed independently of the lifetime of that variable.

An ordinary variable can be named explicitly with a Variable: namespace prefix (§5.2.7).

### Variables on provider drives

The concept of providers and drives is introduced in §3.1, with each provider being able to provide its own namespace drive(s). This allows resources on those drives to be accessed as though they were ordinary variables (§5.2.6). In fact, an ordinary variable is stored on the file system provider drive Variable: (§3.1.5) and can be accessed by its ordinary name or its fully qualified namespace name.

Some namespace variable types are constrained implicitly (§5.3).

## Constrained variables

By default, a variable may designate a value of any type. However, a variable may be constrained to designating values of a given type by specifying that type as a type literal before its name in an assignment or a parameter. For example,

[int]$i = 10 # constrains $i to designating ints only  
$i = "Hello" # error, no conversion to int  
$i = "0x10" # ok, conversion to int  
$i = $true # ok, conversion to int  
  
function F ([int]$p1, [switch]$p2, [regex]$p3) { … }

Any variable belonging to the namespace Env:, Alias:, or to the file system namespace (§2.3.2, §3.1) is constrained implicitly to the type string. Any variable belonging to the namespace Function: (§2.3.2, §3.1) is constrained implicitly to the type scriptblock.

# Conversions

A type conversion is performed when a value of one type is used in a context that requires a different type. If such a conversion happens automatically it is known as implicit conversion. (A common example of this is with some operators that need to convert one or more of the values designated by their operands.) Implicit conversion is permitted provided the sense of the source value is preserved, such as no loss of precision of a number when it is converted.

The cast operator (§7.2.9) allows for explicit conversion.

Conversions are discussed below, with supplementary information being provided as necessary in the description of each operator in §7.

Explicit conversion of a value to the type it already has causes no change to that value or its representation.

The rules for handing conversion when the value of an expression is being bound to a parameter are covered in §6.17.

## Conversion to void

A value of any type can be discarded explicitly by casting it to type void. There is no result.

## Conversion to bool

The rules for converting any value to type bool are as follows:

* A numeric or char value of zero is converted to False; a numeric or char value of non-zero is converted to True.
* A value of null type is converted to False.
* A string of length 0 is converted to False; a string of length > 0 is converted to True.
* A switch parameter with value $true is converted to True, and one with value $false is converted to False.
* All other non-null reference type values are converted to True.

Windows PowerShell: If the type implements IList: If the object's Length > 2, the value is converted to True. If the object's Length is 1 and that first element is not itself an IList, then if that element's value is true, the value is converted to True; otherwise, if the first element’s Count >= 1, the value is converted to True. Otherwise, the value is converted to False.

## Conversion to char

The rules for converting any value to type char are as follows:

* The conversion of a value of type bool, decimal, float, or double is in error.
* A value of null type is converted to the character U+0000.
* An integer type value whose value can be represented in type char has that value; otherwise, the conversion is in error.
* The conversion of a string value having a length other than 1 is in error.
* A string value having a length 1 is converted to a char having that one character's value.
* A numeric type value whose value after rounding of any fractional part can be represented in the destination type has that rounded value; otherwise, the conversion is in error.
* For other reference type values, if the reference type supports such a conversion, that conversion is used; otherwise, the conversion is in error.

## Conversion to integer

The rules for converting any value to type byte, int, or long are as follows:

* The bool value False is converted to zero; the bool value True is converted to 1.
* A char type value whose value can be represented in the destination type has that value; otherwise, the conversion is in error.
* A numeric type value whose value after rounding of any fractional part can be represented in the destination type has that rounded value; otherwise, the conversion is in error.
* A value of null type is converted to zero.
* A string that represents a number is converted as described in §6.16. If after truncation of the fractional part the result can be represented in the destination type the string is well formed and it has the destination type; otherwise, the conversion is in error. If the string does not represent a number, the conversion is in error.
* For other reference type values, if the reference type supports such a conversion, that conversion is used; otherwise, the conversion is in error.

## Conversion to float and double

The rules for converting any value to type float or double are as follows:

* The bool value False is converted to zero; the bool value True is converted to 1.
* A char value is represented is represented exactly.
* A numeric type value is represented exactly, if possible; however, for int, long, and decimal conversions to float, and for long and decimal conversions to double, some of the least significant bits of the integer value may be lost.
* A value of null type is converted to zero.
* A string that represents a number is converted as described in §6.16; otherwise, the conversion is in error.
* For other reference type values, if the reference type supports such a conversion, that conversion is used; otherwise, the conversion is in error.

## Conversion to decimal

The rules for converting any value to type decimal are as follows:

* The bool value False is converted to zero; the bool value True is converted to 1.
* A char type value is represented exactly.
* A numeric type value is represented exactly; however, if that value is too large or too small to fit in the destination type, the conversion is in error.
* A value of null type is converted to zero.
* A string that represents a number is converted as described in §6.16; otherwise, the conversion is in error.
* For other reference type values, if the reference type supports such a conversion, that conversion is used; otherwise, the conversion is in error.
* The scale of the result of a successful conversion is such that the fractional part has no trailing zeros.

## Conversion to object

The value of any type except the null type (4.1.2) can be converted to type object. The value retains its type and representation.

## Conversion to string

The rules for converting any value to type string are as follows:

* The bool value $false is converted to "False"; the bool value $true is converted to "True".
* A char type value is converted to a 1-character string containing that char.
* A numeric type value is converted to a string having the form of a corresponding numeric literal. However, the result has no leading or trailing spaces, no leading plus sign, integers have base 10, and there is no type suffix. For a decimal conversion, the scale is preserved. For values of -∞, +∞, and NaN, the resulting strings are "-Infinity", "Infinity", and "NaN", respectively.
* A value of null type is converted to the empty string.
* For a 1-dimensional array, the result is a string containing the value of each element in that array, from start to end, converted to string, with elements being separated by the current Output Field Separator (§2.3.2.2). For an array having elements that are themselves arrays, only the top-level elements are converted. The string used to represent the value of an element that is an array, is implementation defined. For a multi-dimensional array, it is flattened (§9.12) and then treated as a 1‑dimensional array.
* A value of null type is converted to the empty string.
* A scriptblock type value is converted to a string containing the text of that block without the delimiting { and } characters.
* For an enumeration type value, the result is a string containing the name of each enumeration constant encoded in that value, separated by commas.
* For other reference type values, if the reference type supports such a conversion, that conversion is used; otherwise, the conversion is in error.

Windows PowerShell: The string used to represent the value of an element that is an array has the form System.type[], System.type[,], and so on.

Windows PowerShell: For other reference types, the method ToString is called.

Windows PowerShell: For other enumerable types, the source value is treated like a 1-dimensional array.

## Conversion to array

The rules for converting any value to an array type are as follows:

* The target type may not be a multidimensional array.
* A value of null type is retained as is.
* For a scalar value other than $null or a value of type hashtable, a new 1-element array is created whose value is the scalar after conversion to the target element type.
* For a 1-dimensional array value, a new array of the target type is created, and each element is copied with conversion from the source array to the corresponding element in the target array.
* For a multi-dimensional array value, that array is first flattened (§9.12), and then treated as a 1-dimensional array value.
* A string value is converted to an array of char having the same length with successive characters from the string occupying corresponding positions in the array.

Windows PowerShell: For other enumerable types, a new 1-element array is created whose value is the corresponding element after conversion to the target element type, if such a conversion exists. Otherwise, the conversion is in error.

## Conversion to xml

The object is converted to type string and then into an XML Document object of type xml.

## Conversion to regex

An expression that designates a value of type string may be converted to type regex.

## Conversion to scriptblock

The rules for converting any value to type scriptblock are as follows:

* A string value is treated as the name of a command optionally following by arguments to a call to that command .

## Conversion to enumeration types

The rules for converting any value to an enumeration type are as follows:

* A value of type string that contains one of the named values (with regard for case) for an enumeration type is converted to that named value.
* A value of type string that contains a comma-separated list of named values (with regard for case) for an enumeration type is converted to the bitwise-OR of all those named values.

## Conversion to other reference types

The rules for converting any value to a reference type other than an array type or string are as follows:

* A value of null type is retained as is.
* Otherwise, the behavior is implementation defined.

Windows PowerShell: A number of pieces of machinery come in to play here; these include the possible use of constructors, implicit and explicit conversion operators, and Parse methods for the target type; the use of Convert.ConvertTo; and the ETS conversion mechanism.

## Usual arithmetic conversions

If neither operand designates a value having numeric type, then

* If the left operand designates a value of type bool, the conversion is in error.
* Otherwise, all operands designating the value $null are converted to zero of type int and the process continues with the numeric conversions listed below.
* Otherwise, if the left operand designates a value of type char and the right operand designates a value of type bool, the conversion is in error.
* Otherwise, if the left operand designates a value of type string but does not represent a number (§6.16), the conversion is in error.
* Otherwise, if the right operand designates a value of type string but does not represent a number (§6.16), the conversion is in error.
* Otherwise, all operands designating values of type string are converted to numbers (§6.16), and the process continues with the numeric conversions listed below.
* Otherwise, the conversion is in error.

Numeric conversions:

* If one operand designates a value of type decimal, the value designated by the other operand is converted to that type, if necessary. The result has type decimal.
* Otherwise, if one operand designates a value of type double, the value designated by the other operand is converted to that type, if necessary. The result has type double.
* Otherwise, if one operand designates a value of type float, the values designated by both operands are converted to type double, if necessary. The result has type double.
* Otherwise, if one operand designates a value of type long, the value designated by the other operand value is converted to that type, if necessary. The result has the type first in the sequence long and double that can represent its value.
* Otherwise, the values designated by both operands are converted to type int, if necessary. The result has the first in the sequence int, long, double that can represent its value without truncation.

## Conversion from string to numeric type

Depending on its contents, a string can be converted explicitly or implicitly to a numeric value. Specifically,

* An empty string is converted to the value zero.
* Leading and trailing spaces are ignored; however, a string may not consist of spaces only.
* A string containing only white space and/or line terminators is converted to the value zero.
* One leading + or - sign is permitted.
* An integer number may have a hexadecimal prefix (0x or 0X).
* An optionally signed exponent is permitted.
* Type suffixes and multipliers are not permitted.
* The case-distinct strings "-Infinity", "Infinity", and "NaN" are recognized as the values -∞, +∞, and NaN, respectively.

## Conversion during parameter binding

For information about parameter binding see §8.11.

When the value of an expression is being bound to a parameter, there are extra conversion considerations, as described below:

* If the parameter type is bool or switch (§4.2.5, §8.10.4) and the parameter has no argument, the value of the parameter in the called command is set to $true. If the parameter type is other than bool or switch, a parameter having no argument is in error.
* If the parameter type is switch and the argument value is $null, the parameter value is set to $false.
* If the parameter type is object or is the same as the type of the argument, the argument's value is passed without conversion.
* If the parameter type is not object or scriptblock, an argument having type scriptblock is evaluated and its result is passed as the argument's value. (This is known as delayed script block binding.) If the parameter type is object or scriptblock, an argument having type scriptblock is passed as is.
* If the parameter type is a collection of type T2, and the argument is a scalar of type T1, that scalar is converted to a collection of type T2 containing one element. If necessary, the scalar value is converted to type T2 using the conversion rules of this section.
* If the parameter type is a scalar type other than object and the argument is a collection, the argument is in error.
* If the expected parameter type is a collection of type T2, and the argument is a collection of type T1, the argument is converted to a collection of type T2 having the same length as the argument collection. If necessary, the argument collection element values are converted to type T2 using the conversion rules of this section.
* If the steps above and the conversions specified earlier in this chapter do not suffice, the rules in §6.18 are applied. If those fail, the parameter binding fails.

## .NET Conversion

Windows PowerShell: For an implicit conversion, PowerShell's built-in conversions are tried first. If they cannot resolve the conversion, the .NET custom converters below are tried, in order, from top to bottom. If a conversion is found, but it throws an exception, the conversion has failed.

PSTypeConverter: There are two ways of associating the implementation of the PSTypeConverter class with its target class: through the type configuration file (types.ps1xml) or by applying the System.ComponentModel.TypeConverterAttribute attribute to the target class. Refer to the PowerShell SDK documentation for more information.

TypeConverter: This CLR type provides a unified way of converting types of values to other types, as well as for accessing standard values and sub-properties. The most common type of converter is one that converts to and from a text representation. The type converter for a class is bound to the class with a System.ComponentModel.TypeConverterAttribute. Unless this attribute is overridden, all classes that inherit from this class use the same type converter as the base class. Refer to the PowerShell SDK and the Microsoft .NET framework documentation for more information.

Parse Method: If the source type is string and the destination type has a method called Parse, that method is called to perform the conversion.

Constructors: If the destination type has a constructor taking a single argument whose type is that of the source type, that constructor is called to perform the conversion.

Implicit Cast Operator: If the source type has an implicit cast operator that converts to the destination type, that operator is called to perform the conversion.

Explicit Cast Operator: If the source type has an explicit cast operator that converts to the destination type, that operator is called to perform the conversion. If the destination type has an explicit cast operator that converts from the source type, that operator is called to perform the conversion.

IConvertable: System.Convert.ChangeType is called to perform the conversion.

# Expressions

Syntax:

expression:  
logical-expression

Description:

An expression is a sequence of operators and operands that designates a method, a function, a writable location, or a value; specifies the computation of a value; produces one or more side effects; or performs some combination thereof. For example,

* The literal 123 is an expression that designates the int value 123.
* The expression 1,2,3,4 designates the 4-element array object having the values shown.
* The expression 10.4 \* $a specifies a computation.
* The expression $a++ produces a side effect.
* The expression $a[$i--] = $b[++$j] performs a combination of these things.

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. Examples of unspecified behavior include the following: $i++ + $i, $i + --$i, and $w[$j++] = $v[$j].

An implementation of PowerShell may provide support for user-defined types, and those types may have operations defined on them. All details of such types and operations are implementation defined.

A top-level expression is one that is not part of some larger expression. If a top-level expression contains a side-effect operator the value of that expression is not written to the pipeline; otherwise, it is. See §7.1.1 for a detailed discussion of this.

Ordinarily, an expression that designates a collection (§4) is enumerated into its constituent elements when the value of that expression is used. However, this is not the case when the expression is a cmdlet invocation. For example,

$x = 10,20,30  
$a = $($x; 99) # $a.Length is 4

$x = New-Object 'int[]' 3  
$a = $($x; 99) # equivalent, $a.Length is 4

$a = $(New-Object 'int[]' 3; 99) # $a.Length is 2

In the first two uses of the $(…) operator, the expression designating the collection is the variable $x, which is enumerated resulting in three int values, plus the int 99. However, in the third case, the expression is a direct call to a cmdlet, so the result is not enumerated, and $a is an array of two elements, int[3] and int.

Windows PowerShell: If an operation is not defined by PowerShell, the type of the value designated by the left operand is inspected to see if it has a corresponding op\_<operation> method.

## Primary expressions

Syntax:

primary-expression:  
value  
member-access  
element-access  
invocation-expression  
post-increment-expression  
post-decrement-expression

value:  
parenthesized-expression  
sub-expression  
array-expression  
script-block-expression  
hash-literal-expression  
literal  
type-literal  
variable

### Grouping parentheses

Syntax:

parenthesized-expression:  
( new-linesopt pipeline new-linesopt )

Description:

A parenthesized expression is a primary-expression whose type and value are the same as those of the expression without the parentheses. If the expression designates a variable then the parenthesized expression designates that same variable. For example, $x.m and ($x).m are equivalent.

Grouping parentheses may be used in an expression to document the default precedence and associativity within that expression. They can also be used to override that default precedence and associativity. For example,

4 + 6 \* 2 # 16  
4 + (6 \* 2) # 16 document default precedence  
(4 + 6) \* 2 # 20 override default precedence

Ordinarily, grouping parentheses at the top-most level are redundant. However, that is not always the case. Consider the following example:

2,4,6 # Length 3; values 2,4,6  
(2,4),6 # Length 2; values [object[]],int

In the second case, the parentheses change the semantics, resulting in an array whose two elements are an array of 2 ints and the scalar int 6.

Here's another exception:

23.5/2.4 # pipeline gets 9.79166666666667  
$a = 1234 \* 3.5 # value not written to pipeline  
$a # pipeline gets 4319

In the first and third cases, the value of the result is written to the pipeline. However, although the expression in the second case is evaluated, the result is not written to the pipeline due to the presence of the side-effect operator = at the top level. (Removal of the "$a = " part allows the value to be written, as \* is not a side-effect operator.)

To stop a value of any expression not containing top-level side effects from being written to the pipeline, discard it explicitly, as follows:

[void](23.5/2.4) # value not written to pipeline  
[void]$a # " " "  
$null = $a # " " "  
$a > $null # " " "

To write to the pipeline the value of any expression containing top-level side effects, enclose that expression in parentheses, as follows:

($a = 1234 \* 3.5) # pipeline gets 4319

As such, the grouping parentheses in this case are not redundant.

In the following example, we have variable substitution (§2.3.5.2) taking place in a string literal:

">$($a = -23)<" # value not written to pipeline, get ><  
">$(($a = -23))<" # pipeline gets >-23<

In the first case, the parentheses represent a sub-expression's delimiters not grouping parentheses, and as the top-level expression contains a side-effect operator, the expression's value is not written to the pipeline. Of course, the ">" and "<" characters are still written.) If grouping parenthesis are added—as shown in the second case—writing is enabled.

The following examples each contain top-level side-effect operators:

$a = $b = 0 # value not written to pipeline  
$a = ($b = 0) # value not written to pipeline  
($a = ($b = 0)) # pipeline gets 0

++$a # value not written to pipeline  
(++$b) # pipeline gets 1

$a-- # value not written to pipeline  
($b--) # pipeline gets 1

The use of grouping parentheses around an expression containing no top-level side effects makes those parentheses redundant. For example;

$a # pipeline gets 0  
($a) # no side effect, so () redundant

Consider the following example that has two side effects, neither of which is at the top level:

12.6 + ($a = 10 - ++$b) # pipeline gets 21.6.

The result is written to the pipeline, as the top-level expression has no side effects.

### Member access

Syntax:

member-access: Note no whitespace is allowed between terms in these productions.  
primary-expression . member-name  
primary-expression :: member-name

Description:

The operator . is used to select an instance member from an object, or a key from a Hashtable. The left operand must designate an object, and the right operand must designate an accessible instance member within the type of the object designated by the left operand.

White space is not permitted before or after the . operator.

This operator is left associative.

The operator :: is used to select a static member from a given type. The left operand must designate a type, and the right-hand operand must designate an accessible static member within that type.

White space is not permitted before or after the :: operator.

This operator is left associative.

If the right-hand operand designates a writable location, then the whole expression designates a writable location.

Examples:

$a = 10,20,30  
$a.Length # get instance property

(10,20,30).Length

$property = "Length"  
$a.$property # property name is a variable

$h1 = @{ FirstName = "James"; LastName = "Anderson"; IDNum = 123 }  
$h1.FirstName # designates the key FirstName  
$h1.Keys # gets the collection of keys

[int]::MinValue # get static property  
[double]::PositiveInfinity # get static property  
$property = "MinValue"  
[long]::$property # property name is a variable

foreach ($t in [byte],[int],[long])  
{  
 $t::MaxValue # get static property  
}

### Invocation expressions

Syntax:

invocation-expression: Note no whitespace is allowed between terms in these productions.  
primary-expression . member-name argument-list  
primary-expression :: member-name argument-list

argument-list:  
( argument-expression-listopt new-linesopt )

Description:

An invocation-expression calls the method designated by primary-expression.member-name or primary-expression::member-name. The parentheses in argument-list contain a possibly empty, comma-separated list of expressions, which designate the arguments whose values are passed to the method. Before the method is called, the arguments are evaluated and converted according to the rules of §6, if necessary, to match the types expected by the method. The order of evaluation of primary-expression.member-name, primary-expression::member-name, and the arguments is unspecified.

This operator is left associative.

The type of the result of an invocation-expression is a method designator (§4.5.24).

Examples:

[math]::Sqrt(2.0) # call method with argument 2.0  
[char]::IsUpper("a") # call method  
$b = "abc#$%XYZabc"  
$b.ToUpper() # call instance method

[math]::Sqrt(2) # convert 2 to 2.0 and call method  
[math]::Sqrt(2D) # convert 2D to 2.0 and call method  
[math]::Sqrt($true) # convert $true to 1.0 and call method  
[math]::Sqrt("20") # convert "20" to 20 and call method

$a = [math]::Sqrt # get method descriptor for Sqrt  
$a.Invoke(2.0) # call Sqrt via the descriptor  
$a = [math]::("Sq"+"rt") # get method descriptor for Sqrt  
$a.Invoke(2.0) # call Sqrt via the descriptor  
$a = [char]::ToLower # get method descriptor for ToLower  
$a.Invoke("X") # call ToLower via the descriptor

### Element access

Syntax:

element-access: Note no whitespace is allowed between primary-expression and[.  
primary-expression [ new-linesopt expression new-linesopt ]

Description:

There must not be any white space between primary-expression and the left square bracket ([).

#### Subscripting an array

Description:

Arrays are discussed in detail in §9. If expression is a 1-dimensional array, see §7.1.4.5.

When primary-expression designates a 1-dimensional array A, the operator [] returns the element located at A[0 + expression] after the value of expression has been converted to int. The result has the element type of the array being subscripted. If expression is negative, A[expression] designates the element located at A[A.Length + expression].

When primary-expression designates a 2-dimensional array B, the operator [] returns the element located at B[0 + row,0 + column] after the value of the row and column components of expression (which are specified as a comma-separated list) have been converted to int. The result has the element type of the array being subscripted. Unlike for a 1-dimensional array, negative positions have no special meaning.

When primary-expression designates an array of three or more dimensions, the rules for 2-dimensional arrays apply and the dimension positions are specified as a comma-separated list of values.

If a read access on a non-existing element is attempted, the result is $null. It is an error to write to a non-existing element.

For a multidimensional-array subscript expression, the order of evaluation of the dimension position expressions is unspecified. For example, given a 3-dimensional array $a, the behavior of $a[$i++,$i,++$i] is unspecified.

If expression is an array, see §7.1.4.5.

This operator is left associative.

Examples:

$a = [int[]](10,20,30) # [int[]], Length 3  
$a[1] # returns int 20  
$a[20] # no such position, returns $null  
$a[-1] # returns int 30, i.e., $a[$a.Length-1]  
$a[2] = 5 # changes int 30 to int 5  
$a[20] = 5 # implementation-defined behavior

$a = New-Object 'double[,]' 3,2  
$a[0,0] = 10.5 # changes 0.0 to 10.5  
$a[0,0]++ # changes 10.5 to 10.6

$list = ("red",$true,10),20,(1.2, "yes")  
$list[2][1] # returns string "yes"

$a = @{ A = 10 },@{ B = $true },@{ C = 123.45 }  
$a[1]["B"] # $a[1] is a Hashtable, where B is a key

$a = "red","green"  
$a[1][4] # returns string "n" from string in $a[1]

Windows PowerShell: If a write access to a non-existing element is attempted, an IndexOutOfRange exception is raised.

#### Subscripting a string

Description:

When primary-expression designates a string S, the operator [] returns the character located in the zero-based position indicated by expression, as a char. If expression is greater than or equal to that string's length, the result is $null. If expression is negative, S[expression] designates the element located at S[S.Length + expression].

Examples:

$s = "Hello" # string, Length 5, positions 0-4  
$c = $s[1] # returns "e" as a string  
$c = $s[20] # no such position, returns $null  
$c = $s[-1] # returns "o", i.e., $s[$s.Length-1]

#### Subscripting a Hashtable

Description:

When primary-expression designates a Hashtable, the operator [] returns the value(s) associated with the key(s) designated by expression. The type of expression is not restricted.

When expression is a single key name, the result is the associated value and has that type, unless no such key exists, in which case, the result is $null. If $null is used as the key the behavior is implementation defined. If expression is an array of key names, see §7.1.4.5.

If expression is an array, see §7.1.4.5.

Examples:

$h1 = @{ FirstName = "James"; LastName = "Anderson"; IDNum = 123 }  
$h1['FirstName'] # the value associated with key FirstName  
$h1['BirthDate'] # no such key, returns $null

$h1 = @{ 10 = "James"; 20.5 = "Anderson"; $true = 123 }  
$h1[10] # returns value "James" using key 10  
$h1[20.5] # returns value "Anderson" using key 20.5  
$h1[$true] # returns value 123 using key $true

Windows PowerShell: When expression is a single key name, if $null is used as the only value to subscript a Hashtable, a NullArrayIndex exception is raised.

#### Subscripting an XML document

Description:

When primary-expression designates an object of type xml, expression is converted to string, if necessary, and the operator [] returns the first child element having the name specified by expression. The type of expression must be string. The type of the result is implementation defined. The result can be subscripted to return its first child element. If no child element exists with the name specified by expression, the result is $null. The result does not designate a writable location.

Examples:

$x = [xml]@"  
<Name>  
 <FirstName>Mary</FirstName>  
 <LastName>King</LastName>  
</Name>  
"@

$x['Name'] # refers to the element Name  
$x['Name']['FirstName'] # refers to the element FirstName within Name  
$x['FirstName'] # No such child element at the top level, result is $null

Windows PowerShell: The type of the result is System.Xml.XmlElement or System.String.

#### Generating array slices

When primary-expression designates an object of a type that is enumerable (§4) or a Hashtable, and expression is a 1-dimensional array, the result is an array slice (§9.9) containing the elements of primary-expression designated by the elements of expression.

In the case of a Hashtable, the array slice contains the associated values to the keys provided, unless no such key exists, in which case, the corresponding element is $null. If $null is used as any key name the behavior is implementation defined.

Examples:

$a = [int[]](30,40,50,60,70,80,90)  
$a[1,3,5] # slice has Length 3, value 40,60,80  
++$a[1,3,5][1] # preincrement 60 in array 40,60,80  
$a[,5] # slice with Length 1  
$a[@()] # slice with Length 0  
$a[-1..-3] # slice with Length 0, value 90,80,70

$a = New-Object 'int[,]' 3,2  
$a[0,0] = 10; $a[0,1] = 20; $a[1,0] = 30  
$a[1,1] = 40; $a[2,0] = 50; $a[2,1] = 60  
$a[(0,1),(1,0)] # slice with Length 2, value 20,30, parens needed

$h1 = @{ FirstName = "James"; LastName = "Anderson"; IDNum = 123 }  
$h1['FirstName'] # the value associated with key FirstName  
$h1['BirthDate'] # no such key, returns $null

$h1['FirstName','IDNum'] # returns [object[]], Length 2 (James/123)  
$h1['FirstName','xxx'] # returns [object[]], Length 2 (James/$null)  
$h1[$null,'IDNum'] # returns [object[]], Length 1 (123)

Windows PowerShell: When expression is a collection of two or more key names, if $null is used as any key name that key is ignored and has no corresponding element in the resulting array.

### Postfix increment and decrement operators

Syntax:

post-increment-expression:  
primary-expression ++

post-decrement-expression:  
primary-expression dashdash

dashdash:  
dash dash

Description:

The primary-expression must designate a writable location having a value of numeric type (§4) or the value $null. If the value designated by the operand is $null, that value is converted to type int and value zero before the operator is evaluated. [Note: The type of the value designated by primary-expression may change when the result is stored. See §7.11 for a discussion of type change via assignment. end note]

The result produced by the postfix ++ operator is the value designated by the operand. After that result is obtained, the value designated by the operand is incremented by 1 of the appropriate type. The type of the result of expression E++ is the same as for the result of the expression E + 1 (§7.7).

The result produced by the postfix -- operator is the value designated by the operand. After that result is obtained, the value designated by the operand is decremented by 1 of the appropriate type. The type of the result of expression E-- is the same as for the result of the expression E - 1 (§7.7).

These operators are left associative.

Examples:

$i = 0 # $i = 0  
$i++ # $i is incremented by 1  
$j = $i-- # $j takes on the value of $i before the decrement

$a = 1,2,3  
$b = 9,8,7  
$i = 0  
$j = 1  
$b[$j--] = $a[$i++] # $b[1] takes on the value of $a[0], then $j is  
 # decremented, $i incremented

$i = 2147483647 # $i holds a value of type int  
$i++ # $i now holds a value of type double because  
 # 2147483648 is too big to fit in type int

[int]$k = 0 # $k is constrained to int  
$k = [int]::MaxValue # $k is set to 2147483647  
$k++ # 2147483648 is too big to fit, imp-def bahavior

$x = $null # target is unconstrained, $null goes to [int]0  
$x++ # value treated as int, 0->1

### $(…) operator

Syntax:

sub-expression:  
$( new-linesopt statement-listopt new-linesopt )

Description:

If statement-list is omitted, the result is $null. Otherwise, statement-list is evaluated. Any objects written to the pipeline as part of the evaluation are collected in an unconstrained 1-dimensional array, in order. If the array of collected objects is empty, the result is $null. If the array of collected objects contains a single element, the result is that element; otherwise, the result is the unconstrained 1-dimensional array of collected results.

Examples:

$j = 20  
$($i = 10) # pipeline gets nothing  
$(($i = 10)) # pipeline gets int 10  
$($i = 10; $j) # pipeline gets int 20  
$(($i = 10); $j) # pipeline gets [object[]](10,20)  
$(($i = 10); ++$j) # pipeline gets int 10  
$(($i = 10); (++$j)) # pipeline gets [object[]](10,22)  
$($i = 10; ++$j) # pipeline gets nothing  
$(2,4,6) # pipeline gets [object[]](2,4,6)

### @(…) operator

Syntax:

array-expression:  
@( new-linesopt statement-listopt new-linesopt )

Description:

If statement-list is omitted, the result is an unconstrained 1-dimensional array of length zero. Otherwise, statement-list is evaluated, and any objects written to the pipeline as part of the evaluation are collected in an unconstrained 1-dimensional array, in order. The result is the (possibly empty) unconstrained 1-dimensional array.

Examples:

$j = 20  
@($i = 10) # 10 not written to pipeline, result is array of 0  
@(($i = 10)) # pipeline gets 10, result is array of 1  
@($i = 10; $j) # 10 not written to pipeline, result is array of 1  
@(($i = 10); $j) # pipeline gets 10, result is array of 2  
@(($i = 10); ++$j) # pipeline gets 10, result is array of 1  
@(($i = 10); (++$j)) # pipeline gets both values, result is array of 2  
@($i = 10; ++$j) # pipeline gets nothing, result is array of 0

$a = @(2,4,6) # result is array of 3  
@($a) # result is the same array of 3  
@(@($a)) # result is the same array of 3

### Script block expression

Syntax:

script-block-expression:  
{ new-linesopt script-block new-linesopt }

script-block:  
param-blockopt script-block-bodyopt

script-block-body:  
named-block-list  
statement-list

Description:

param-block is described in §8.10.8. named-block-list is described in §8.10.6.

A script block is an unnamed block of statements that can be used as a single unit. Script blocks can be used to invoke a block of code as if it was a single command, or they can be assigned to variables that can be executed.

The named-block-list or statement-list is executed and the type and value(s) of the result are the type and value(s) of the results of those statement sets.

A script-block-expression has type scriptblock (§4.3.7).

If param-block is omitted, any arguments passed to the script block are available via $args (§8.10.1).

During parameter binding, a script block can be passed either as a script block object or as the result after the script block has been evaluated. See §6.17 for further information.

### Hash literal expression

Syntax:

hash-literal-expression:  
@{ new-linesopt hash-literal-bodyopt new-linesopt }

hash-literal-body:  
hash-entry  
hash-literal-body statement-terminators hash-entry

hash-entry:  
key-expression = new-linesopt statement

key-expression:  
simple-name  
unary-expression

statement-terminators:  
statement-terminator  
statement-terminators statement-terminator

statement-terminator:  
;  
new-line-character

Description:

A hash-literal-expression is used to create a Hashtable (§10) of zero or more elements each of which is a key/value pair.

The key may have any type except the null type. The associated values may have any type, including the null type, and each of those values may be any expression that designates the desired value, including $null.

The ordering of the key/value pairs is not significant.

Examples:

$h1 = @{ FirstName = "James"; LastName = "Anderson"; IDNum = 123 }  
$last = "Anderson"; $IDNum = 120  
$h2 = @{ FirstName = "James"; LastName = $last; IDNum = $IDNum + 3 }  
$h3 = @{ }  
$h4 = @{ 10 = "James"; 20.5 = "Anderson"; $true = 123 }

which creates two Hashtables, $h1 and $h2, each containing three key/value pairs, and a third, $h3, that is empty. Hashtable $h4 has keys of various types.

### Type literal expression

Syntax:

type-literal:  
[ type-spec ]

type-spec:  
array-type-name dimensionopt ]  
generic-type-name generic-type-arguments ]  
type-name

dimension:  
,  
*dimension* ,

generic-type-arguments:  
type-spec  
generic-type-arguments , type-spec

array-type-name:  
type-name [

generic-type-name:  
type-name [

Description:

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.

Type literals are used in a number of contexts:

* Specifying an explicit conversion (§6, §7.2.9)
* Creating a type-constrained array (§9.4)
* Accessing the static members of an object (§7.1.2)
* Specifying a type constraint on a variable (§5.3) or a function parameter (§8.10.2)

Examples:

Examples of type literals are [int], [object[], and [int[,,]]. A generic stack type (§4.4) that is specialized to hold strings might be written as [Stack[string]], and a generic dictionary type that is specialized to hold int keys with associated string values might be written as [Dictionary[int,string]].

Windows PowerShell: The type of a type-literal is System.Type.

Windows PowerShell: The complete name for the type Stack[string] suggested above is System.Collections.Generic.Stack[int]. The complete name for the type Dictionary[int,string] suggested above is System.Collections.Generic.Dictionary[int,string].

## Unary operators

Syntax:

unary-expression:  
primary-expression  
expression-with-unary-operator

expression-with-unary-operator:  
, new-linesopt unary-expression  
-not new-linesopt unary-expression  
! new-linesopt unary-expression  
-bnot new-linesopt unary-expression  
+ new-linesopt unary-expression  
dash new-linesopt unary-expression  
pre-increment-expression  
pre-decrement-expression  
cast-expression  
-split new-linesopt unary-expression  
-join new-linesopt unary-expression

dash:  
- (U+002D)  
EnDash character (U+2013)  
EmDash character (U+2014)  
Horizontal bar character (U+2015)

pre-increment-expression:  
++ new-linesopt unary-expression

pre-decrement-expression:  
dashdash new-linesopt unary-expression

cast-expression:  
type-literal unary-expression

dashdash:  
dash dash

### Unary comma operator

Description:

This operator creates an unconstrained 1-dimensional array having one element, whose type and value are that of unary-expression.

This operator is right associative.

Examples:

$a = ,10 # create an unconstrained array of 1 element, $a[0],  
 # which has type int

$a = ,(10,"red") # create an unconstrained array of 1 element, $a[0],  
 # which is an unconstrained array of 2 elements,  
 # $a[0][0] an int, and $a[0][1] a string

$a = ,,10 # create an unconstrained array of 1 element, which is  
 # an unconstrained array of 1 element, which is an int  
 # $a[0][0] is the int. Contrast this with @(@(10))

### Logical NOT

Description:

The operator -not converts the value designated by unary-expression to type bool (§6.2), if necessary, and produces a result of that type. If unary-expression's value is True, the result is False, and vice versa. The operator ! is an alternate spelling for -not.

This operator is right associative.

Examples:

-not $true # False  
-not -not $false # False  
-not 0 # True  
-not 1.23 # False  
!"xyz" # False

### Bitwise NOT

Description:

The operator -bnot converts the value designated by unary-expression to an integer type (§6.4), if necessary. If the converted value can be represented in type int then that is the result type. Else, if the converted value can be represented in type long then that is the result type. Otherwise, the expression is ill formed. The resulting value is the ones-complement of the converted value.

This operator is right associative.

Examples:

-bnot $true # int with value 0xFFFFFFFE  
-bnot 10 # int with value 0xFFFFFFF5  
-bnot 2147483648.1 # long with value 0xFFFFFFFF7FFFFFFF  
-bnot $null # int with value 0xFFFFFFFF  
-bnot "0xabc" # int with value 0xFFFFF543

### Unary plus

Description:

An expression of the form +unary-expression is treated as if it were written as 0 + unary-expression (§7.7). [Note: The integer literal 0 has type int. end note]

This operator is right associative.

Examples:

+123L # type long, value 123  
+0.12340D # type decimal, value 0.12340  
+"0xabc" # type int, value 2748

### Unary minus

Description:

An expression of the form -unary-expression is treated as if it were written as 0 - unary-expression (§7.7). [Note: The integer literal 0 has type int. end note]

This operator is right associative.

Examples:

-$true # type int, value -1  
-123L # type long, value -123  
-0.12340D # type decimal, value -0.12340

### Prefix increment and decrement operators

Description:

The unary-expression must designate a writable location having a value of numeric type (§4) or the value $null. If the value designated by its unary-expression is $null, unary-expression's value is converted to type int and value zero before the operator is evaluated. [Note: The type of the value designated by unary-expression may change when the result is stored. See §7.11 for a discussion of type change via assignment. end note]

For the prefix ++ operator, the value of unary-expression is incremented by 1 of the appropriate type. The result is the new value after incrementing has taken place. The expression ++E is equivalent to E += 1 (§7.11.2).

For the prefix -- operator, the value of unary-expression is decremented by 1 of the appropriate type. The result is the new value after decrementing has taken place. The expression --E is equivalent to E -= 1 (§7.11.2).

These operators are right associative.

Examples:

$i = 0 # $i = 0  
$++i # $i is incremented by 1  
$j = --$i # $i is decremented then $j takes on the value of $i

$a = 1,2,3  
$b = 9,8,7  
$i = 0;  
$j = 1  
$b[--$j] = $a[++$i] # $j is # decremented, $i incremented, then $b[0]  
 # takes on the value of $a[1]

$i = 2147483647 # $i holds a value of type int  
++$i # $i now holds a value of type double because  
 # 2147483648 is too big to fit in type int

[int]$k = 0 # $k is constrained to int  
$k = [int]::MinValue # $k is set to -2147483648  
$--k # -2147483649 is too small to fit, imp-def behavior

$x = $null # target is unconstrained, $null goes to [int]0  
$--x # value treated as int, 0->-1

### The unary -join operator

Description:

The unary -join operator produces a string that is the concatenation of the value of one or more objects designated by unary-expression. (A separator can be inserted by using the binary version of this operator (§7.8.4.4).)

unary-expression can be a scalar value or a collection.

Examples:

-join (10, 20, 30) # result is "102030"  
-join (123, $false, 19.34e17) # result is "123False1.934E+18"  
-join 12345 # result is "12345"  
-join $null # result is ""

### The unary -split operator

Description:

The unary -split operator splits one or more strings designated by unary-expression, returning their subparts in a constrained 1-dimensional array of string. It treats any contiguous group of white space characters as the delimiter between successive subparts. (An explicit delimiter string can be specified by using the binary version of this operator (§7.8.4.5).) This operator has two variants (§7.8).

The delimiter text is not included in the resulting strings. Leading and trailing white space in the input string is ignored. An input string that is empty or contains white space only results in an array of 1 string, which is empty.

unary-expression can designate a scalar value or an array of strings.

Examples:

-split " red`tblue`ngreen " # 3 strings: "red", "blue", "green"  
-split ("yes no", "up down") # 4 strings: "yes", "no", "up", "down"  
-split " " # 1 (empty) string

### Cast operator

Description:

This operator converts explicitly (§6) the value designated by unary-expression to the type designated by type-literal. If type-literal is other than void, the type of the result is the named type, and the value is the value after conversion. If type-literal is void, no object is written to the pipeline and there is no result.

When an expression of any type is cast to that same type, the resulting type and value is the unary-expression's type and value.

This operator is right associative.

Examples:

[bool]-10 # a bool with value True  
[int]-10.70D # a decimal with value -10  
[int]10.7 # an int with value 11  
[long]"+2.3e+3" # a long with value 2300  
[char[]]"Hello" # an array of 5 char with values H, e, l, l, and o.

## Binary comma operator

Syntax:

array-literal-expression:  
unary-expression  
unary-expression , new-linesopt array-literal-expression

Description:

The binary comma operator creates a 1-dimensional array whose elements are the values designated by its operands, in lexical order. The array has unconstrained type.

Examples:

2,4,6 # Length 3; values 2,4,6  
(2,4),6 # Length 2; values [object[]],int  
(2,4,6),12,(2..4) # Length 3; [object[]],int,[object[]]  
2,4,6,"red",$null,$true # Length 6

[Note: The addition of grouping parentheses to certain binary comma expressions does not document the default precedence; instead, it changes the result. end note]

## Range operator

Syntax:

range-expression:  
array-literal-expression  
range-expression .. new-linesopt array-literal-expression

Description:

A range-expression creates an unconstrained 1-dimensional array whose elements are the values of the int sequence specified by the range bounds. The values designated by the operands are converted to int, if necessary (§6.4). The operand designating the lower value after conversion is the lower bound, while the operand designating the higher value after conversion is the upper bound. Both bounds may be the same, in which case, the resulting array has length 1. If the left operand designates the lower bound, the sequence is in ascending order. If the left operand designates the upper bound, the sequence is in descending order.

[Note: Conceptually, this operator is a shortcut for the corresponding binary comma operator sequence. For example, the range 5..8 can also be generated using 5,6,7,8. However, if an ascending or descending sequence is needed without having an array, an implementation may avoid generating an actual array. For example, in foreach ($i in 1..5) { … }, no array need be created. end note]

A range-expression can be used to specify an array slice (§9.9).

Examples:

1..10 # ascending range 1..10  
-500..-495 # descending range -500..-495  
16..16 # seqeunce of 1

$x = 1.5  
$x..5.40D # ascending range 2..5

$true..3 # ascending range 1..3  
-2..$null # ascending range -2..0  
"0xf".."0xa" # descending range 15..10

## Format operator

Syntax:

format-expression:  
range-expression  
format-expression format-operator new-linesopt range-expression

format-operator:  
dash f

dash:  
- (U+002D)  
EnDash character (U+2013)  
EmDash character (U+2014)  
Horizontal bar character (U+2015)

Description:

A format-expression formats one or more values designated by range-expression according to a format specification string designated by format-expression. The positions of the values designated by range-expression are numbered starting at zero and increasing in lexical order. The result has type string.

A format specification string may contain zero or more format specifications each having the following form:

{N [ ,M ][ : FormatString ]}

N represents a (required) range-expression value position, M represents the (optional) minimum display width, and FormatString indicates the (optional) format. If the width of a formatted value exceeds the specified width, the width is increased accordingly. Values whose positions are not referenced in FormatString are ignored after being evaluated for any side effects. If N refers to a non-existent position, the behavior is implementation defined. Value of type $null and void are formatted as empty strings. Arrays are formatted as for sub-expression (§7.1.6). To include the characters "{" and "}" in a format specification without their being interpreted as format delimiters, write them as "{{" and "}}", respectively.

For a complete definition of format specifications, see the type System.IFormattable in Ecma Technical Report TR/84.

Examples:

$i = 10; $j = 12  
"{2} <= {0} + {1}`n" -f $i,$j,($i+$j) # 22 <= 10 + 12  
">{0,3}<" -f 5 # > 5<  
">{0,-3}<" -f 5 # >5 <  
">{0,3:000}<" -f 5 # >005<  
">{0,5:0.00}<" -f 5.0 # > 5.00<  
">{0:C}<" -f 1234567.888 # >$1,234,567.89<   
">{0:C}<" -f -1234.56 # >($1,234.56)<  
">{0,12:e2}<" -f 123.456e2 # > 1.23e+004<   
">{0,-12:p}<" -f -0.252 # >-25.20 % <

$format = ">{0:x8}<"  
$format -f 123455 # >0001e23f<

Windows PowerShell: In a format specification if N refers to a non-existent position, a FormatError is raised.

## Multiplicative operators

Syntax:

multiplicative-expression:  
format-expression  
multiplicative-expression \* new-linesopt format-expression  
multiplicative-expression / new-linesopt format-expression  
multiplicative-expression % new-linesopt format-expression

### Multiplication

Description:

The result of the multiplication operator \* is the product of the values designated by the two operands after the usual arithmetic conversions (§6.15) have been applied.

This operator is left associative.

Examples:

12 \* -10L # long result -120  
-10.300D \* 12 # decimal result -123.600  
10.6 \* 12 # double result 127.2  
12 \* "0xabc" # int result 32976

### String replication

Description:

When the left operand designates a string the binary \* operator creates a new string that contains the one designated by the left operand replicated the number of times designated by the value of the right operand as converted to integer type (§6.4).

This operator is left associative.

Examples:

"red" \* "3" # string replicated 3 times  
"red" \* 4 # string replicated 4 times  
"red" \* 0 # results in an empty string  
"red" \* 2.3450D # string replicated twice  
"red" \* 2.7 # string replicated 3 times

### Array replication

Description:

When the left operand designates an array the binary \* operator creates a new unconstrained 1‑dimensional array that contains the value designated by the left operand replicated the number of times designated by the value of the right operand as converted to integer type (§6.4). A replication count of zero results in an array of length 1. If the left operand designates a multidimensional array, it is flattened (§9.12) before being used.

This operator is left associative.

Examples:

$a = [int[]](10,20) # [int[]], Length 2\*1  
$a \* "3" # [object[]], Length 2\*3  
$a \* 4 # [object[]], Length 2\*4  
$a \* 0 # [object[]], Length 2\*0  
$a \* 2.3450D # [object[]], Length 2\*2  
$a \* 2.7 # [object[]], Length 2\*3  
(New-Object 'float[,]' 2,3) \* 2 # [object[]], Length 2\*2

### Division

Description:

The result of the division operator / is the quotient when the value designated by the left operand is divided by the value designated by the right operand after the usual arithmetic conversions (§6.15) have been applied.

If an attempt is made to perform integer or decimal division by zero, an implementation-defined terminating error is raised.

This operator is left associative.

Examples:

10/-10 # int result -1.2  
12/-10 # double result -1.2  
12/-10D # decimal result 1.2  
12/10.6 # double result 1.13207547169811  
12/"0xabc" # double result 0.00436681222707424

Windows PowerShell: If an attempt is made to perform integer or decimal division by zero, a RuntimeException exception is raised.

### Remainder

Description:

The result of the remainder operator % is the remainder when the value designated by the left operand is divided by the value designated by the right operand after the usual arithmetic conversions (§6.15) have been applied.

If an attempt is made to perform integer or decimal division by zero, an implementation-defined terminating error is raised.

Examples:

10 % 3 # int result 1  
10.0 % 0.3 # double result 0.1  
10.00D % "0x4" # decimal result 2.00

Windows PowerShell: If an attempt is made to perform integer or decimal division by zero, a RuntimeException exception is raised.

## Additive operators

Syntax:

additive-expression:  
multiplicative-expression  
additive-expression + new-linesopt multiplicative-expression  
additive-expression dash new-linesopt multiplicative-expression

### Addition

Description:

The result of the addition operator + is the sum of the values designated by the two operands after the usual arithmetic conversions (§6.15) have been applied.

This operator is left associative.

Examples:

12 + -10L # long result 2  
-10.300D + 12 # decimal result 1.700  
10.6 + 12 # double result 22.6  
12 + "0xabc" # int result 2760

### String concatentaion

Description:

When the left operand designates a string the binary + operator creates a new string that contains the value designated by the left operand followed immediately by the value(s) designated by the right operand as converted to type string (§6.8).

This operator is left associative.

Examples:

"red" + "blue" # "redblue"  
"red" + "123" # "red123"  
"red" + 123 # "red123"  
"red" + 123.456e+5 # "red12345600"  
"red" + (20,30,40) # "red20 30 40"

### Array concatenation

Description:

When the left operand designates an array the binary + operator creates a new unconstrained 1‑dimensional array that contains the elements designated by the left operand followed immediately by the value(s) designated by the right operand. Multidimensional arrays present in either operand are flattened (§9.12) before being used.

This operator is left associative.

Examples:

$a = [int[]](10,20) # [int[]], Length 2  
$a + "red" # [object[]], Length 3  
$a + 12.5,$true # [object[]], Length 4  
$a + (New-Object 'float[,]' 2,3) # [object[]], Length 8  
(New-Object 'float[,]' 2,3) + $a # [object[]], Length 8

### Hashtable concatenation

Description:

When both operands designate Hashtables the binary + operator creates a new Hashtable that contains the elements designated by the left operand followed immediately by the elements designated by the right operand.

If the Hashtables contain the same key, an implementation-defined terminating error is raised..

This operator is left associative.

Examples:

$h1 = @{ FirstName = "James"; LastName = "Anderson" }  
$h2 = @{ Dept = "Personnel" }  
$h3 = $h1 + $h2 # new Hashtable, Count = 3

Windows PowerShell: If the Hashtables contain the same key, an exception of type BadOperatorArgument is raised.

### Subtraction

Description:

The result of the subtraction operator - is the difference when the value designated by the right operand is subtracted from the value designated by the left operand after the usual arithmetic conversions (§6.15) have been applied.

This operator is left associative.

Examples:

12 - -10L # long result 2c  
 -10.300D - 12 # decimal result -22.300  
 10.6 - 12 # double result -1.4  
 12 - "0xabc" # int result -2736

## Comparison operators

Syntax:

comparison-operator: one of  
dash as dash ccontains dash ceq  
dash cge dash cgt dash cle  
dash clike dash clt dash cmatch  
dash cne dash cnotcontains dash cnotlike  
dash cnotmatch dash contains dash creplace  
dash csplit dash eq dash ge  
dash gt dash icontains dash ieq  
dash ige dash igt dash ile  
dash ilike dash ilt dash imatch  
dash ine dash inotcontains dash inotlike  
dash inotmatch dash ireplace dash is  
dash isnot dash isplit dash join  
dash le dash like dash lt  
dash match dash ne dash notcontains  
dash notlike dash notmatch dash replace  
dash split

dash:  
- (U+002D)  
EnDash character (U+2013)  
EmDash character (U+2014)  
Horizontal bar character (U+2015)

Description:

The type of the value designated by the left operand determines how the value designated by the right operand is converted (§6), if necessary, before the comparison is done.

Some comparison-operators (written here as -op) have two variants, one that is case sensitive (-cop), and one that is not (-iop). The -op version is equivalent to -iop. Case sensitivity is meaningful only with comparisons of values of type string. In non-string comparison contexts, the two variants behave the same.

These operators are left associative.

### Equality and relational operators

Description:

There are two equality operators: equality (-eq) and inequality (-ne); and four relational operators: less-than (-lt), less-than-or-equal-to (le), greater-than (-gt), and greater-than-or-equal-to (le). Each of these has two variants (§7.8).

For two strings to compare equal, they must have the same length and contents, and letter case, if appropriate.

If the value designated by the left operand is not a collection, the result has type bool. Otherwise, the result is a possibly empty unconstrained 1-dimensional array containing the elements of the collection that test True when compared to the value designated by the right operand.

Examples:

10 -eq "010" # True, int comparison  
"010" -eq 10 # False, string comparison  
"RED" -eq "Red" # True, case-insensitive comparison  
"RED" -ceq "Red" # False, case-sensitive comparison  
"ab" -lt "abc" # True

10,20,30,20,10 -ne 20 # 10,30,10, Length 3  
10,20,30,20,10 -eq 40 # Length 0  
10,20,30,20,10 -ne 40 # 10,20,30,20,10, Length 5  
10,20,30,20,10 -gt 25 # 30, Length 1  
0,1,30 -ne $true # 0,30, Length 2  
0,"00" -eq "0" # 0 (int), Length 1

### Containment operators

Description:

There are two containment operators: contains (-contains) and does-not-contain (‑notcontains). Each of these has two variants (§7.8).

The containment operators return a result of type bool that indicates whether the value designated by the right operand occurs (or does not occur) at least once in the elements of the array designated by the left operand. For the purposes of these operators, a scalar value is treated as an array of one element, allowing it to be the left operand.

Examples:

10,20,30,20,10 -contains 20 # True  
10,20,30,20,10 -contains 42.9 # False  
10,20,30 -contains "10" # True  
"010",20,30 -contains 10 # False  
10,20,30,20,10 -notcontains 15 # True  
"Red",20,30 -ccontains "RED" # False

### Type testing and conversion operators

Description:

The type operator -is tests whether the value designated by the left operand has the type, or is derived from a type that has the type, designated by the right operand. The right operand must designate a type or a value that can be converted to a type (such as a string that names a type). The type of the result is bool. The type operator -isnot returns the logical negation of the corresponding -is form.

The type operator -as attempts to convert the value designated by the left operand to the type designated by the right operand. The right operand must designate a type or a value that can be converted to a type (such as a string that names a type). If the conversion fails, $null is returned; otherwise, the converted value is returned and the return type of that result is the runtime type of the converted value.

Examples:

$a = 10 # value 10 has type int  
$a -is [int] # True

$t = [int]  
$a -isnot $t # False  
$a -is "int" # True  
$a -isnot [double] # True

$x = [int[]](10,20)  
$x -is [int[]] # True

$a = "abcd" # string is derived from object  
$a -is [object] # True

$x = [double]  
foreach ($t in [int],$x,[decimal],"string")  
{  
 $b = (10.60D -as $t) \* 2 # results in int 22, double 21.2  
} # decimal 21.20, and string "10.6010.60"

### Pattern matching and text manipulation operators

#### The -like and -notlike operators

Description:

If the left operand does not designate a collection, the result has type bool. Otherwise, the result is a possibly empty unconstrained 1-dimensional array containing the elements of the collection that test True when compared to the value designated by the right operand. The right operand may designate a string that contains wildcard expressions (§3.15). These operators have two variants (§7.8).

Examples:

"Hello" -like "h\*" # True, starts with h  
"Hello" -clike "h\*" # False, does not start with lowercase h  
"Hello" -like "\*l\*" # True, has an l in it somewhere  
"Hello" -like "??l" # False, no length match

"-abc" -like "[-xz]\*" # True, - is not a range separator  
"#$%^&" -notlike "\*[A-Za-z]" # True, does not end with alphabetic character   
"He" -like "h[aeiou]?\*" # False, need at least 3 characters  
"When" -like "\*[?]" # False, ? is not a wildcard character  
"When?" -like "\*[?]" # True, ? is not a wildcard character

"abc","abbcde","abcgh" -like "abc\*" # object[2], values "abc" and "abcgh"

#### The -match and -notmatch operators

Description:

If the left operand does not designate a collection, the result has type bool and if that result is $true, the elements of the Hashtable $matches are set to the strings that match (or do-not-match) the value designated by the right operand. Otherwise, the result is a possibly empty unconstrained 1-dimensional array containing the elements of the collection that test True when compared to the value designated by the right operand, and $matches is not set. The right operand may designate a string that contains regular expressions (§3.16), in which case, it is referred to as a pattern. These operators have two variants (§7.8).

These operators support submatches (§7.8.4.6).

Examples:

"Hello" -match ".l" # True, $matches key/value is 0/"el"  
"Hello" -match '^h.\*o$' # True, $matches key/value is 0/"Hello"  
"Hello" -cmatch '^h.\*o$' # False, $matches not set  
"abc^ef" -match ".\^e" # True, $matches key/value is 0/"c^e"

"abc" -notmatch "[A-Za-z]" # False  
"abc" -match "[^A-Za-z]" # False  
"He" -match "h[aeiou]." # False, need at least 3 characters  
"abc","abbcde","abcgh" -match "abc.\*" # Length is 2, values "abc", "abcgh"

#### The -replace operator

Description:

The -replace operator allows text replacement in one or more strings designated by the left operand using the values designated by the right operand. This operator has two variants (§7.8). The right operand has one of the following forms:

* The string to be located, which may contain regular expressions (§3.16). In this case, the replacement string is implicitly "".
* An array of 2 objects containing the string to be located, followed by the replacement string.

If the left operand designates a string, the result has type string. If the left operand designates a 1‑dimensional array of string, the result is an unconstrained 1-dimensional array—whose length is the same as for left operand's array—containing the input strings after replacement has completed.

This operator supports submatches (§7.8.4.6).

Examples:

"Analogous","an apple" -replace "a","\*" # "\*n\*logous","\*n \*pple"  
"Analogous" -creplace "[aeiou]","?" # "An?l?g??s"  
"Analogous","an apple" -replace '^a',"%%A" # "%%Analogous","%%An apple"  
"Analogous" -replace "[aeiou]",'$&$&' # "AAnaaloogoouus"

#### The binary -join operator

Description:

The binary -join operator produces a string that is the concatenation of the value of one or more objects designated by the left operand after having been converted to string (§6.7), if necessary. The string designated by the right operand is used to separate the (possibly empty) values in the resulting string.

The left operand can be a scalar value or a collection.

Examples:

(10, 20, 30) -join "|" # result is "10|20|30"  
12345 -join "," # result is "12345", no separator needed  
($null,$null) -join "<->" # result is "<->", two zero-length values

#### The binary -split operator

Description:

The binary -split operator splits one or more strings designated by the left operand, returning their subparts in a constrained 1-dimensional array of string. This operator has two variants (§7.8). The left operand can designate a scalar value or an array of strings. The right operand has one of the following forms:

* A delimiter string
* An array of 2 objects containing a delimiter string followed by a numeric split count
* An array of 3 objects containing a delimiter string, a numeric split count, and an options string
* A script block
* An array of 2 objects containing a script block followed by a numeric split count

The delimiter string may contain regular expressions (§3.16). It is used to locate subparts with the input strings. The delimiter is not included in the resulting strings. If the left operand designates an empty string, that results in an empty string element. If the delimiter string is an empty string, it is found at every character position in the input strings.

By default, all subparts of the input strings are placed into the result as separate elements; however, the split count can be used to modify this behavior. If that count is negative, zero, or greater than or equal to the number of subparts in an input string, each subpart goes into a separate element. If that count is less than the number of subparts in the input string, there are count elements in the result, with the final element containing all of the subparts beyond the first count - 1 subparts.

An options string contains zero or more option names with each adjacent pair separated by a comma. Leading, trailing, and embedded white space is ignored. Option names may be in any order and are case-sensitive.

If an options string contains the option name SimpleMatch, it may also contain the option name IgnoreCase. If an options string contains the option name RegexMatch or it does not contain either RegexMatch or SimpleMatch, it may contain any option name except SimpleMatch. However, it must not contain both Multiline and Singleline.

Here is the set of option names:

| **Option** | **Description** |
| --- | --- |
| CultureInvariant | Ignores cultural differences in language when evaluating the delimiter. |
| ExplicitCapture | Ignores non-named match groups so that only explicit capture groups are returned in the result list. |
| IgnoreCase | Force case-insensitive matching, even if -csplit is used. |
| IgnorePatternWhitespace | Ignores unescaped white space and comments marked with the number sign (#). |
| Multiline | This mode recognizes the start and end of lines and strings. The default mode is Singleline. |
| RegexMatch | Use regular expression matching to evaluate the delimiter. This is the default. |
| SimpleMatch | Use simple string comparison when evaluating the delimiter. |
| Singleline | This mode recognizes only the start and end of strings. It is the default mode. |

The script block (§7.1.8) specifies the rules for determining the delimiter, and must evaluate to type bool.

Examples:

"one,forty two,," -split "," # 5 strings: "one" "forty two" "" ""

"abc","de" -split "" # 9 strings: "" "a" "b" "c" "" "" "d"  
 # "e" ""

"ab,cd","1,5,7,8" -split ",", 2 # 4 strings: "ab" "cd" "1" "5,7,8"

"10X20x30" -csplit "X", 0, "SimpleMatch" # 2 strings: "10" "20x30"

"analogous" -split "[AEIOU]", 0, "RegexMatch, IgnoreCase"  
 # 6 strings: "" "n" "l" "g" "" "s"

"analogous" -split { $\_ -eq "a" -or $\_ -eq "o" }, 4  
 # 4 strings: "" "n" "l" "gous"

#### Submatches

The pattern being matched by -match, -notmatch, and -replace may contain subparts—called submatches—delimited by parentheses. Consider the following example:

"red" -match "red"

The result is $true and key 0 of $matches contains "red", that part of the string designated by the left operand that exactly matched the pattern designated by the right operand.

In the following example, the whole pattern is a submatch:

"red" -match "(red)"

As before, key 0 contains "red"; however, key 1 also contains "red", which is that part of the string designated by the left operand that exactly matched the submatch.

Consider the following, more complex, pattern:

"red" -match "((r)e)(d)"

This pattern allows submatches of "r", "re", "d", or "red".

Again, key 0 contains "red". Key 1 contains "re", key 2 contains "r", and key 3 contains "d". The key/value pairs are in matching order from left-to-right in the pattern, with longer string matches preceding shorter ones.

In the case of -replace, the replacement text can access the submatches via names of the form $n, where the first match is $1, the second is $3, and so on. For example,

"Monday morning" -replace '(Monday|Tuesday)   
 (morning|afternoon|evening)','the $2 of $1'

The resulting string is "the morning of Monday".

Windows PowerShell: Instead of having keys in $matches be zero-based indexes, submatches can be named using the form ?<name>. For example, "((r)e)(d)" can be written with three named submatches, m1, m2, and m3, as follows: "(?<m1>(?<m2>r)e)(?<m3>d)".

## Bitwise operators

Syntax:

bitwise-expression:  
comparison-expression  
bitwise-expression -band new-linesopt comparison-expression  
bitwise-expression -bor new-linesopt comparison-expression  
bitwise-expression -bxor new-linesopt comparison-expression

Description:

The bitwise AND operator -band, the bitwise OR operator -bor, and the bitwise XOR operator -bxor convert the values designated by their operands to integer types, if necessary, using the usual arithmetic conversions (§6.15). After conversion, if both values have type int that is the type of the result. Otherwise, if both values have type long, that is the type of the result. If one value has type int and the other has type long, the type of the result is long. Otherwise, the expression is ill formed. The result is the bitwise AND, bitwise OR, or bitwise XOR, respectively, of the possibly converted operand values.

These operators are left associative. They are commutative if neither operand contains a side effect.

Examples:

0x0F0F -band 0xFE # int with value 0xE  
0x0F0F -band 0xFEL # long with value 0xE  
0x0F0F -band 14.6 # long with value 0xF

0x0F0F -bor 0xFE # int with value 0xFFF  
0x0F0F -bor 0xFEL # long with value 0xFFF  
0x0F0F -bor 14.40D # long with value 0xF0F

0x0F0F -bxor 0xFE # int with value 0xFF1  
0x0F0F -bxor 0xFEL # long with value 0xFF1  
0x0F0F -bxor 14.40D # long with value 0xF01  
0x0F0F -bxor 14.6 # long with value 0xF00

## Logical operators

Syntax:

logical-expression:  
bitwise-expression  
logical-expression -and new-linesopt bitwise-expression  
logical-expression -or new-linesopt bitwise-expression  
logical-expression -xor new-linesopt bitwise-expression

Description:

The logical AND operator -and converts the values designated by its operands to bool, if necessary (§6.2). The result is the logical AND of the possibly converted operand values, and has type bool. If the left operand evaluates to False the right operand is not evaluated.

The logical OR operator -or converts the values designated by its operands to bool, if necessary (§6.2). The result is the logical OR of the possibly converted operand values, and has type bool. If the left operand evaluates to True the right operand is not evaluated.

The logical XOR operator -xor converts the values designated by its operands to bool (§6.2). The result is the logical XOR of the possibly converted operand values, and has type bool.

These operators are left associative.

Examples:

$j = 10  
$k = 20  
($j -gt 5) -and (++$k -lt 15) # True -and False -> False  
($j -gt 5) -and ($k -le 21) # True -and True -> True  
($j++ -gt 5) -and ($j -le 10) # True -and False -> False  
($j -eq 5) -and (++$k -gt 15) # False -and True -> False

$j = 10  
$k = 20  
($j++ -gt 5) -or (++$k -lt 15) # True -or False -> True  
($j -eq 10) -or ($k -gt 15) # False -or True -> True  
($j -eq 10) -or (++$k -le 20) # False -or False -> False

$j = 10  
$k = 20  
($j++ -gt 5) -xor (++$k -lt 15) # True -xor False -> True  
($j -eq 10) -xor ($k -gt 15) # False -xor True -> True  
($j -gt 10) -xor (++$k -le 25) # True -xor True -> False

## Assignment operators

Syntax:

assignment-expression:  
expression assignment-operator statement

assignment-operator: one of  
= dash = += \*= /= %=

Description:

An assignment operator stores a value in the writable location designated by expression. For a discussion of assignment-operator = see §7.11.1. For a discussion of all other assignment-operators see §7.11.2.

An assignment expression has the value designated by expression after the assignment has taken place; however, that assignment expression does not itself designate a writable location. If expression is type-constrained (§5.3), the type used in that constraint is the type of the result; otherwise, the type of the result is the type after the usual arithmetic conversions (§6.15) have been applied.

This operator is right associative.

### Simple assignment

Description:

In simple assignment (=), the value designated by statement replaces the value stored in the writable location designated by expression. However, if expression designates a non-existent key in a Hashtable, that key is added to the Hashtable with an associated value of the value designated by statement.

As shown by the grammar, expression may designate a comma-separated list of writable locations. This is known as multiple assignment. statement designates a list of one or more comma-separated values. The commas in either operand list are part of the multiple-assignment syntax and do not represent the binary comma operator. Values are taken from the list designated by statement, in lexical order, and stored in the corresponding writable location designated by expression. If the list designated by statement has fewer values than there are expression writable locations, the excess locations take on the value $null. If the list designated by statement has more values than there are expression writable locations, all but the right-most expression location take on the corresponding statement value and the right-most expression location becomes an unconstrained 1-dimensional array with all the remaining statement values as elements.

For statements that have values (§8.1.2), statement can be a statement.

Examples:

$a = 20; $b = $a + 12L # $b has type long, value 22  
$hypot = [Math]::Sqrt(3\*3 + 4\*4) # type double, value 5  
$a = $b = $c = 10.20D # all have type decimal, value 10.20  
$a = (10,20,30),(1,2) # type [object[]], Length 2  
[int]$x = 10.6 # type int, value 11  
[long]$x = "0xabc" # type long, value 0xabc  
$a = [float] # value type literal [float]

$i,$j,$k = 10,"red",$true # $i is 10, $j is "red", $k is True  
$i,$j = 10,"red",$true # $i is 10, $j is [object[]], Length 2  
$i,$j = (10,"red"),$true # $i is [object[]], Length 2, $j is True  
$i,$j,$k = 10 # $i is 10, $j is $null, $k is $null

$h = @{}  
[int] $h.Lower, [int] $h.Upper = -split "10 100"

$h1 = @{ FirstName = "James"; LastName = "Anderson"; IDNum = 123 } $h1.Dept = "Finance" # adds element Finance  
$h1["City"] = "New York" # adds element City

[int]$Variable:v = 123.456 # v takes on the value 123  
${E:output.txt} = "a" # write text to the given file  
$Env:MyPath = "x:\data\file.txt" # define the environment variable  
$Function:F = { param ($a, $b) "Hello there, $a, $b" }  
F 10 "red" # define and invoke a function  
function Demo { "Hi there from inside Demo" }  
$Alias:A = "Demo" # create alias for function Demo  
A # invoke function Demo via the alias

### Compound assignment

Description:

A compound assignment has the form E1 op= E2, and is equivalent to the simple assignment expression E1 = E1 op (E2) except that in the compound assignment case the expression E1 is evaluated only once. If expression is type-constrained (§5.3), the type used in that constraint is the type of the result; otherwise, the type of the result is determined by op. (For \*=, see §7.6.1, §7.6.2, §7.6.3; for /=, see §7.6.4; for %=, see §7.6.5; for +=, see §7.7.1, §7.7.2, §7.7.3; for -=, see §7.7.5.)

[Note: An operand designating an unconstrained value of numeric type may have its type changed by an assignment operator when the result is stored. end note]

Examples:

$a = 1234; $a \*= (3 + 2) # type is int, value is 1234 \* (3 + 2)  
$b = 10,20,30 # $b[1] has type int, value 20   
$b[1] /= 6 # $b[1] has type double, value 3.33…

$i = 0  
$b = 10,20,30  
$b[++$i] += 2 # side effect evaluated only once

[int]$Variable:v = 10 # v takes on the value 10  
$Variable:v -= 3 # 3 is subtracted from v

${E:output.txt} = "a" # write text to the given file  
${E:output.txt} += "b" # append text to the file giving ab  
${E:output.txt} \*= 4 # replicate ab 4 times giving abababab

## Redirection operators

Syntax:

pipeline:  
assignment-expression  
expression redirectionsopt pipeline-tailopt  
command pipeline-tailopt

redirections:  
redirection  
redirections redirection

redirection:  
2>&1  
1>&2  
file-redirection-operator redirected-file-name

redirected-file-name:  
command-argument  
primary-expression

Description:

The redirection operator > takes the standard output from the pipeline and redirects it to the location designated by redirected-file-name, overwriting that location's current contents.

The redirection operator >> takes the standard output from the pipeline and redirects it to the location designated by redirected-file-name, appending to that location's current contents, if any. If that location does not exist, it is created.

The redirection operator 2> takes the error output from the pipeline and redirects it to the location designated by redirected-file-name, overwriting that location's current contents.

The redirection operator 2>> takes the error output from the pipeline and redirects it to the location designated by redirected-file-name, appending to that location's current contents, if any. If that location does not exist, it is created.

The redirection operator 2>&1 writes error output to the same location as the standard output.

The redirection operators 1>&2 and < are reserved for future use.

If on output the value of redirected-file-name is $null, the output is discarded.

Ordinarily, the value of an expression containing a top-level side effect is not written to the pipeline unless that expression is enclosed in a pair of parentheses. However, if such an expression is the left operand of an operator that redirects standard output, the value is written.

Examples:

$i = 200 # pipeline gets nothing  
$i # pipeline gets result  
$i > output1.txt # result redirected to named file  
++$i >> output1.txt # result appended to named file  
type file1.txt 2> error1.txt # error output redirected to named file  
type file2.txt 2>> error1.txt # error output appended to named file

# Statements

## Statement blocks and lists

Syntax:

statement-block:  
new-linesopt { statement-listopt new-linesopt }

statement-list:  
statement  
statement-list statement

statement:  
if-statement  
labelopt labeled-statement  
function-statement  
flow-control-statement statement-terminator  
trap-statement  
try-statement  
data-statement  
pipeline statement-terminator

statement-terminator:  
;  
new-line-character

Description:

A statement specifies some sort of action that is to be performed. Unless indicated otherwise within this clause, statements are executed in lexical order.

A statement-block allows a set of statements to be grouped into a single syntactic unit.

### Labeled statements

Syntax:

labeled-statement:  
switch-statement  
foreach-statement  
for-statement  
while-statement  
do-statement

Description:

An iteration statement (§8.4) or a switch statement (§8.6) may optionally be preceded immediately by one statement label, label. A statement label is used as the optional target of a break (§8.5.1) or continue (§8.5.2) statement. However, a label does not alter the flow of control.

White space is not permitted between the colon (:) and the token that follows it.

Examples:

:go\_here while ($j -le 100)  
{  
 # …  
}

:labelA  
 for ($i = 1; $i -le 5; ++$i)  
 {  
:labelB  
 for ($j = 1; $j -le 3; ++$j)  
 {  
:labelC  
 for ($k = 1; $k -le 2; ++$k)  
 {  
 # …  
 }  
 }  
 }

### Statement values

The value of a statement is the cumulative set of values that it writes to the pipeline. If the statement writes a single scalar value, that is the value of the statement. If the statement writes multiple values, the value of the statement is that set of values stored in elements of an unconstrained 1-dimensional array, in the order in which they were written. Consider the following example:

$v = for ($i = 10; $i -le 5; ++$i) { }

There are no iterations of the loop and nothing is written to the pipeline. The value of the statement is $null.

$v = for ($i = 1; $i -le 5; ++$i) { }

Although the loop iterates five times nothing is written to the pipeline. The value of the statement is $null.

$v = for ($i = 1; $i -le 5; ++$i) { $i }

The loop iterates five times each time writing to the pipeline the int value $i. The value of the statement is object[] of Length 5.

$v = for ($i = 1; $i -le 5; ) { ++$i }

Although the loop iterates five times nothing is written to the pipeline. The value of the statement is $null.

$v = for ($i = 1; $i -le 5; ) { (++$i) }

The loop iterates five times with each value being written to the pipeline. The value of the statement is object[] of Length 5.

$i = 1; $v = while ($i++ -lt 2) { $i }

The loop iterates once. The value of the statement is the int with value 2.

Here are some other examples:

# if $count is not currently defined then define it with int value 10  
$count = if ($count -eq $null) { 10 } else { $count }

$i = 1  
$v = while ($i -le 5)  
{  
 $i # $i is written to the pipeline  
 if ($i -band 1)  
 {

"odd" # conditionally written to the pipeline

}

++$i # not written to the pipeline

}  
# $v is object[], Length 8, value 1,"odd",2,3,"odd",4,5,"odd"

## Pipeline statements

Syntax:

pipeline:  
assignment-expression  
expression redirectionsopt pipeline-tailopt  
command pipeline-tailopt

assignment-expression:  
expression assignment-operator statement

pipeline-tail:  
| new-linesopt command  
| new-linesopt command pipeline-tail

command:  
command-name command-elementsopt  
command-invocation-operator command-moduleopt command-name-expr command-elementsopt

command-invocation-operator: one of  
& .

command-module:  
primary-expression

command-name:  
generic-token  
generic-token-with-subexpr

generic-token-with-subexpr:  
No whitespace is allowed between ) and command-name.  
generic-token-with-subexpr-start statement-listopt ) command-name

command-name-expr:  
command-name  
primary-expression

command-elements:  
command-element  
command-elements command-element

command-element:  
command-parameter  
command-argument  
redirection

command-argument:  
command-name-expr

Description:

redirections is discussed in §7.12; assignment-expression is discussed in §7.11; and the command-invocation-operator dot (.) is discussed in §3.5.5. For a discussion of argument-to-parameter mapping in command invocations, see §8.11.

The first command in a pipeline is an expression or a command invocation. Typically, a command invocation begins with a command-name, which is usually a bare identifier. command-elements represents the argument list to the command. A newline or n unescaped semicolon terminates a pipeline.

A command invocation consists of the command's name followed by zero or more arguments. The rules governing arguments are as follows:

* An argument that is not an expression, but which contains arbitrary text without unescaped white space, is treated as though it were double quoted. Letter case is preserved.
* Variable substitution and sub-expression expansion (§2.3.5.2) takes place inside expandable-string-literals and expandable-here-string-literals.
* Text inside quotes allows leading, trailing, and embedded white space to be included in the argument's value. [Note: The presence of whitespace in a quoted argument does not turn a single argument into multiple arguments. end note]
* Putting parentheses around an argument causes that expression to be evaluated with the result being passed instead of the text of the original expression.
* To pass an argument that looks like a switch parameter (§2.3.4) but is not intended as such, enclose that argument in quotes.
* When specifying an argument that matches a parameter having the [switch] type constraint (§8.10.4), the presence of the argument name on its own causes that parameter to be set to $true. However, the parameter's value can be set explicitly by appending a suffix to the argument. For example, given a type constrained parameter p, an argument of -p:$true sets p to True, while -p:$false sets p to False.
* An argument of -- indicates that all arguments following it are to be passed in their actual form as though double quotes were placed around them.

The order of evaluation of arguments is unspecified.

For information about parameter binding see §8.11. For information about name lookup see §3.8.

Once argument processing has been completed, the command is invoked. If the invoked command terminates normally (§8.5.4), control reverts to the point in the script or function immediately following the command invocation. For a description of the behavior on abnormal termination see break (§8.5.1), continue (§8.5.2), throw (§8.5.3), exit (§8.5.5), try (§8.7), and trap (§8.8).

Ordinarily, a command is invoked by using its name followed by any arguments. However, the command-invocation operator, &, can be used. If the command name contains unescaped white space, it must be quoted and invoked with this operator. As a script block has no name, it too must be invoked with this operator. For example, the following invocations of a command call Get-Factorial are equivalent:

Get-Factorial 5  
& Get-Factorial 5  
& "Get-Factorial" 5

Direct and indirect recursive function calls are permitted. For example,

function Get-Power([int]$x, [int]$y)  
{  
 if ($y -gt 0) { return $x \* (Get-Power $x (--$y)) }  
 else { return 1 }  
}

Examples:

New-Object 'int[,]' 3,2  
New-Object -ArgumentList 3,2 -TypeName 'int[,]'

dir e:\PowerShell\Scripts\\*statement\*.ps1 | Foreach-Object {$\_.Length}

dir e:\PowerShell\Scripts\\*.ps1 | Select-String -List "catch" | Format-Table  
 path,linenumber –AutoSize

## The if statement

Syntax:

if-statement:  
if new-linesopt ( new-linesopt pipeline new-linesopt ) statement-block  
 elseif-clausesopt else-clauseopt

elseif-clauses:  
elseif-clause  
elseif-clauses elseif-clause

elseif-clause:  
new-linesopt elseif new-linesopt ( new-linesopt pipeline new-linesopt ) statement-block

else-clause:  
new-linesopt else statement-block

Description:

The pipeline controlling expressions must have type bool or be implicitly convertible to that type. The else-clause is optional. There may be zero or more elseif-clauses.

If the top-level pipeline tests True, then its statement-block is executed and execution of the statement terminates. Otherwise, if an elseif-clause is present, if its pipeline tests True, then its statement-block is executed and execution of the statement terminates. Otherwise, if an else-clause is present, its statement-block is executed.

Examples:

$grade = 92  
if ($grade -ge 90) { "Grade A" }  
elseif ($grade -ge 80) { "Grade B" }  
elseif ($grade -ge 70) { "Grade C" }  
elseif ($grade -ge 60) { "Grade D" }  
else { "Grade F" }

## Iteration statements

### The while statement

Syntax:

while-statement:  
while new-linesopt ( new-linesopt while-condition new-linesopt ) statement-block

while-condition:  
new-linesopt pipeline

Description:

The controlling expression while-condition must have type bool or be implicitly convertible to that type. The loop body, which consists of statement-block, is executed repeatedly until the controlling expression tests False. The controlling expression is evaluated before each execution of the loop body.

Examples:

$i = 1  
while ($i -le 5) # loop 5 times  
{  
 "{0,1}`t{1,2}" -f $i, ($i\*$i)  
 ++$i  
}

### The do statement

Syntax:

do-statement:  
do statement-block new-linesopt while new-linesopt ( while-condition new-linesopt )  
do statement-block new-linesopt until new-linesopt ( while-condition new-linesopt )

while-condition:  
new-linesopt pipeline

Description:

The controlling expression while-condition must have type bool or be implicitly convertible to that type. In the while form, the loop body, which consists of statement-block, is executed repeatedly while the controlling expression tests True. In the until form, the loop body is executed repeatedly until the controlling expression tests True. The controlling expression is evaluated after each execution of the loop body.

Examples:

$i = 1  
do  
{  
 "{0,1}`t{1,2}" -f $i, ($i\*$i)  
}  
while (++$i -le 5) # loop 5 times

$i = 1  
do  
{  
 "{0,1}`t{1,2}" -f $i, ($i\*$i)  
}  
until (++$i -gt 5) # loop 5 times

### The for statement

Syntax:

for-statement:  
for new-linesopt (  
 new-linesopt for-initializeropt ;  
 new-linesopt for-conditionopt ;  
 new-linesopt for-iteratoropt  
 new-linesopt ) statement-block  
for new-linesopt (  
 new-linesopt for-initializeropt ;  
 new-linesopt for-conditionopt  
 new-linesopt ) statement-block  
for new-linesopt (  
 new-linesopt for-initializeropt  
 new-linesopt ) statement-block

for-initializer:  
pipeline

for-condition:  
pipeline

for-iterator:  
pipeline

Description:

The controlling expression for-condition must have type bool or be implicitly convertible to that type. The loop body, which consists of statement-block, is executed repeatedly while the controlling expression tests True. The controlling expression is evaluated before each execution of the loop body.

Expression for-initializer is evaluated before the first evaluation of the controlling expression. Expression for-initializer is evaluated for its side effects only; any value it produces is discarded and is not written to the pipeline.

Expression for-iterator is evaluated after each execution of the loop body. Expression for-iterator is evaluated for its side effects only; any value it produces is discarded and is not written to the pipeline.

If expression for-condition is omitted, the controlling expression tests True.

Examples:

for ($i = 5; $i -ge 1; --$i) # loop 5 times  
{  
 "{0,1}`t{1,2}" -f $i, ($i\*$i)  
}

$i = 5  
for (;$i -ge 1;) # equivalent behavior  
{  
 "{0,1}`t{1,2}" -f $i, ($i\*$i)  
 --$i  
}

### The foreach statement

Syntax:

foreach-statement:  
foreach new-linesopt ( new-linesopt variable new-linesopt in new-linesopt pipeline  
 new-linesopt ) statement-block

Description:

The loop body, which consists of statement-block, is executed for each element designated by the variable variable in the collection designated by pipeline. The scope of variable is not limited to the foreach statement. As such, it retains its final value after the loop body has finished executing. If pipeline designates a scalar (including the value $null) instead of a collection, that scalar as treated as a collection of one element.

Every foreach statement has its own enumerator, $foreach (§2.3.2.2, §4.5.16), which exists only while that loop is executing.

The objects produced by pipeline are collected before statement-block begins to execute. However, with the ForEach-Object cmdlet (§13.12), statement-block is executed on each object as it is produced.

Examples:

$a = 10,53,16,-43  
foreach ($e in $a)  
{  
 …  
}  
$e # the int value -43

foreach ($e in -5..5)  
{  
 …  
}

foreach ($t in [byte],[int],[long])  
{  
 $t::MaxValue # get static property  
}

foreach ($f in dir \*.txt)  
{  
 …  
}

$h1 = @{ FirstName = "James"; LastName = "Anderson"; IDNum = 123 }  
foreach ($e in $h1.Keys)  
{  
 "Key is " + $e + ", Value is " + $h1[$e]  
}

## Flow control statements

Syntax:

flow-control-statement:  
break label-expressionopt  
continue label-expressionopt  
throw pipelineopt  
return pipelineopt  
exit pipelineopt

label-expression:  
simple-name  
unary-expression

Description:

A flow-control statement causes an unconditional transfer of control to some other location.

### The break statement

Description:

A break statement with a label-expression is referred to as a labeled break statement. A break statement without a label-expression is referred to as an unlabeled break statement.

Outside a trap statement, an unlabeled break statement directly within an iteration statement (§8.4) terminates execution of that smallest enclosing iteration statement. An unlabeled break statement directly within a switch statement (§8.6) terminates pattern matching for the current switch's switch-condition. See (§8.8) for details of using break from within a trap statement.

An iteration statement or a switch statement may optionally be preceded immediately by one statement label (§8.1.1).Such a statement label may be used as the target of a labeled break statement, in which case, that statement terminates execution of the targeted enclosing iteration statement.

A labeled break need not be resolved in any local scope; the search for a matching label may continue up the calling stack even across script and function-call boundaries. If no matching label is found, the current command invocation is terminated.

The name of the label designated by label-expression need not have a constant value.

If label-expression is a unary-expression, it is converted to a string.

Examples:

$i = 1  
while ($true) # infinite loop  
{  
 if ($i \* $i -gt 100)  
 {  
 break # break out of current while loop  
 }  
 ++$i  
}

$lab = "go\_here"  
:go\_here  
 for ($i = 1; ; ++$i)  
 {  
 if ($i \* $i -gt 50)  
 {  
 break $lab # use a string value as target  
 }  
 }

:labelA  
 for ($i = 1; $i -le 2; $i++)  
 {

:labelB  
 for ($j = 1; $j -le 2; $j++)  
 {

:labelC  
 for ($k = 1; $k -le 3; $k++)  
 {  
 if (…) { break labelA }  
 }  
 }  
 }

### The continue statement

Description:

A continue statement with a label-expression is referred to as a labeled continue statement. A continue statement without a label-expression is referred to as an unlabeled continue statement.

The use of continue from within a trap statement is discussed in §8.8.

An unlabeled continue statement within a loop terminates execution of the current loop and transfers control to the closing brace of the smallest enclosing iteration statement (§8.4). An unlabeled continue statement within a switch terminates execution of the current switch iteration and transfers control to the smallest enclosing switch's switch-condition (§8.6).

An iteration statement or a switch statement (§8.6) may optionally be preceded immediately by one statement label (§8.1.1).Such a statement label may be used as the target of an enclosed labeled continue statement, in which case, that statement terminates execution of the current loop or switch iteration, and transfers control to the targeted enclosing iteration or switch statement label.

A labeled continue need not be resolved in any local scope; the search for a matching label may continue up the calling stack even across script and function-call boundaries. If no matching label is found, the current command invocation is terminated.

The name of the label designated by label-expression need not have a constant value.

If label-expression is a unary-expression, it is converted to a string.

Examples:

$i = 1  
while (…)  
{  
 …  
 if (…)  
 {  
 continue # start next iteration of current loop  
 }  
 …  
}

$lab = "go\_here"  
:go\_here  
 for (…; …; …)  
 {  
 if (…)  
 {  
 continue $lab # start next iteration of labeled loop  
 }  
 }

:labelA  
 for ($i = 1; $i -le 2; $i++)  
 {

:labelB  
 for ($j = 1; $j -le 2; $j++)  
 {

:labelC  
 for ($k = 1; $k -le 3; $k++)  
 {  
 if (…) { continue labelB }  
 }  
 }  
 }

### The throw statement

Description:

An exception is a way of handling a system- or application-level error condition. The throw statement raises an exception. (See §8.7 for a discussion of exception handling.)

If pipeline is omitted and the throw statement is not in a catch-clause, the behavior is implementation defined. If pipeline is present and the throw statement is in a catch-clause, the exception that was caught by that catch-clause is re-thrown after any finally-clause associated with the catch-clause is executed.

If pipeline is present, the type of the exception thrown is implementation defined.

When an exception is thrown, control is transferred to the first catch clause in an enclosing try statement that can handle the exception. The location at which the exception is thrown initially is called the throw point. Once an exception is thrown the steps described in §8.7 are followed repeatedly until a catch clause that matches the exception is found or none can be found.

Examples:

throw  
throw 100  
throw "No such record in file"

Windows PowerShell: If pipeline is omitted and the throw statement is not from within a catch-clause, the text "ScriptHalted" is written to the pipeline, and the type of the exception raised is System.Management.Automation.RuntimeException.

Windows PowerShell: If pipeline is present, the exception raised is wrapped in an object of type System.Management.Automation.RuntimeException, which includes information about the exception as a System.Management.Automation.ErrorRecord object (accessible via $\_).

Example 1: throw 123 results in an exception of type RuntimeException. From within the catch block, $\_.TargetObject contains the object wrapped inside, in this case, a System.Int32 with value 123.

Example 2: throw "xxx" results in an exception of type RuntimeException. From within the catch block, $\_.TargetObject contains the object wrapped inside, in this case, a System.String with value "xxx".

Example 3: throw 10,20 results in an exception of type RuntimeException. From within the catch block, $\_.TargetObject contains the object wrapped inside, in this case, a System.Object[], an unconstrained array of two elements with the System.Int32 values 10 and 20.

### The return statement

Description:

The return statement writes to the pipeline the value(s) designated by pipeline, if any, and returns control to the function or script's caller. A function or script may have zero or more return statements.

If execution reaches the closing brace of a function an implied return without pipeline is assumed.

The return statement is a bit of "syntactic sugar" to allow programmers to express themselves as they can in other languages; however, the value returned from a function or script is actually all of the values written to the pipeline by that function or script plus any value(s) specified by pipeline. If only a scalar value is written to the pipeline, its type is the type of the value returned; otherwise, the return type is an unconstrained 1-dimensional array containing all the values written to the pipeline.

Examples:

function Get-Factorial ($v)  
{  
 if ($v -eq 1)  
 {  
 return 1 # return is not optional  
 }  
  
 return $v \* (Get-Factorial ($v - 1)) # return is optional  
}

The caller to Get-Factorial gets back an int.

function Test  
{  
 "text1" # "text1" is written to the pipeline  
# …  
 "text2" # "text2" is written to the pipeline  
# …  
 return 123 # 123 is written to the pipeline  
}

The caller to Test gets back an unconstrained 1-dimensional array of three elements.

### The exit statement

Description:

The exit statement terminates the current script and returns control and an exit code to the host environment or the calling script. If pipeline is provided, the value it designates is converted to int, if necessary. If no such conversion exists, or if pipeline is omitted, the int value zero is returned.

Examples:

exit $count # terminate the script with some accumulated count

## The switch statement

Syntax:

switch-statement:  
switch new-linesopt switch-parametersopt switch-condition switch-body

switch-parameters:  
switch-parameter  
switch-parameters switch-parameter

switch-parameter:  
-regex  
-wildcard  
-exact  
-casesensitive

switch-condition:  
( new-linesopt pipeline new-linesopt )  
-file new-linesopt switch-filename

switch-filename:  
command-argument  
primary-expression

switch-body:  
new-linesopt { switch-clauses new-linesopt }

switch-clauses:  
switch-clause  
switch-clauses switch-clause

switch-clause:  
new-linesopt switch-clause-condition statement-block

switch-clause-condition:  
command-argument  
primary-expression

Description:

If switch-condition designates a single value, control is passed to one or more matching pattern statement blocks. If no patterns match, some default action can be taken.

A switch must contain one or more switch-clauses, each starting with a pattern (a non-default switch clause), or the keyword default (a default switch clause). A switch must contain zero or one default switch clauses, and zero or more non-default switch clauses. Switch clauses may be written in any order.

Multiple patterns may have the same value. A pattern need not be a literal, and a switch may have patterns with different types.

If the value of switch-condition matches a pattern value, that pattern's statement-block is executed. If multiple pattern values match the value of switch-condition, each matching pattern's statement-block is executed, in lexical order, unless any of those statement-blocks contains a break statement (§8.5.1).

If the value of switch-condition does not match any pattern value, if a default switch clause exists, its statement-block is executed; otherwise, pattern matching for that switch-condition is terminated.

Switches may be nested, with each switch having its own set of switch clauses. In such instances, a switch clause belongs to the innermost switch currently in scope.

On entry to each statement-block, $\_ is automatically assigned the value of the switch-condition that caused control to go to that statement-block. $\_ is also available in that statement-block's switch-clause-condition.

Matching of non-strings is done by testing for equality (§7.8.1).

If the matching involves strings, by default, the comparison is case-insensitive. The presence of the switch-parameter -casesensitive makes the comparison case-sensitive.

A pattern may contain wildcard characters (§3.15), in which case, wildcard string comparisons are performed, but only if the switch-parameter -wildcard is present. By default, the comparison is case-insensitive.

A pattern may contain a regular expression (§3.16), in which case, regular expression string comparisons are performed, but only if the switch-parameter -regex is present. By default, the comparison is case-insensitive. If -regex is present and a pattern is matched, $matches is defined in the switch-clause statement-block for that pattern.

A switch-parameter may be abbreviated; any distinct leading part of a parameter may be used. For example, ‑regex, ‑rege, ‑reg, ‑re, and ‑r are equivalent.

If conflicting switch-parameters are specified, the lexically final one prevails. The presence of ‑exact disables -regex and -wildcard; it has no affect on ‑case, however.

If a pattern is a script-block-expression, that block is evaluated and the result is converted to bool, if necessary. If the result has the value $true, the corresponding statement-block is executed; otherwise, it is not.

If switch-condition designates multiple values, the switch is applied to each value in lexical order using the rules described above for a switch-condition that designates a single value. Every switch statement has its own enumerator, $switch (§2.3.2.2, §4.5.16), which exists only while that switch is executing.

A switch statement may have a label, and it may contain labeled and unlabeled break (§8.5.1) and continue (§8.5.2) statements.

If switch-condition is -file switch-filename, instead of iterating over the values in an expression, the switch iterates over the values in the file designated by switch-filename.The file is read a line at a time with each line comprising a value. Line terminator characters are not included in the values.

Examples:

$s = "ABC def`nghi`tjkl`fmno @#$"  
$charCount = 0; $pageCount = 0; $lineCount = 0; $otherCount = 0  
for ($i = 0; $i -lt $s.Length; ++$i)  
{  
 ++$charCount  
 switch ($s[$i])  
 {  
 "`n" { ++$lineCount }  
 "`f" { ++$pageCount }  
 "`t" { }  
 " " { }  
 default { ++$otherCount }  
 }  
}

switch -wildcard ("abc")  
{  
 a\* { "a\*, $\_" }  
 ?B? { "?B? , $\_" }  
 default { "default, $\_" }  
}

switch -regex -casesensitive ("abc")  
{  
 ^a\* { "a\*" }  
 ^A\* { "A\*" }  
}

switch (0,1,19,20,21)  
{  
 { $\_ -lt 20 } { "-lt 20" }  
 { $\_ -band 1 } { "Odd" }  
 { $\_ -eq 19 } { "-eq 19" }  
 default { "default" }  
}

## The try/finally statement

Syntax:

try-statement:  
try statement-block catch-clauses  
try statement-block finally-clause  
try statement-block catch-clauses finally-clause

catch-clauses:  
catch-clause  
catch-clauses catch-clause

catch-clause:  
new-linesopt catch catch-type-listopt statement-block

catch-type-list:  
new-linesopt type-literal  
catch-type-list new-linesopt , new-linesopt type-literal

finally-clause:  
new-linesopt finally statement-block

Description:

The try statement provides a mechanism for catching exceptions that occur during execution of a block. The try statement also provides the ability to specify a block of code that is always executed when control leaves the try statement. The process of raising an exception via the throw statement is described in §8.5.3.

A try block is the statement-block associated with the try statement. A catch block is the statement-block associated with a catch-clause. A finally block is the statement-block associated with a finally-clause.

A catch-clause without a catch-type-list is called a general catch clause.

Each catch-clause is an exception handler, and a catch-clause whose catch-type-list contains the type of the raised exception is a matching catch clause. A general catch clause matches all exception types.

Although catch-clauses and finally-clause are optional, at least one of them must be present.

The processing of a thrown exception consists of evaluating the following steps repeatedly until a catch clause that matches the exception is found.

* In the current scope, each try statement that encloses the throw point is examined. For each try statement S, starting with the innermost try statement and ending with the outermost try statement, the following steps are evaluated:
* If the try block of S encloses the throw point and if S has one or more catch clauses, the catch clauses are examined in lexical order to locate a suitable handler for the exception. The first catch clause that specifies the exception type or a base type of the exception type is considered a match. A general catch clause is considered a match for any exception type. If a matching catch clause is located, the exception processing is completed by transferring control to the block of that catch clause. Within a matching catch clause, the variable $\_ contains a description of the current exception.
* Otherwise, if the try block or a catch block of S encloses the throw point and if S has a finally block, control is transferred to the finally block. If the finally block throws another exception, processing of the current exception is terminated. Otherwise, when control reaches the end of the finally block, processing of the current exception is continued.
* If an exception handler was not located in the current scope, the steps above are then repeated for the enclosing scope with a throw point corresponding to the statement from which the current scope was invoked.
* If the exception processing ends up terminating all scopes, indicating that no handler exists for the exception, then the behavior is unspecified.

To prevent unreachable catch clauses in a try block, a catch clause may not specify an exception type that is equal to or derived from a type that was specified in an earlier catch clause within that same try block.

The statements of a finally block are always executed when control leaves a try statement. This is true whether the control transfer occurs as a result of normal execution, as a result of executing a break, continue, or return statement, or as a result of an exception being thrown out of the try statement.

If an exception is thrown during execution of a finally block, the exception is thrown out to the next enclosing try statement. If another exception was in the process of being handled, that exception is lost. The process of generating an exception is further discussed in the description of the throw statement.

try statements can co-exist with trap statements; see §8.8 for details.

Examples:

$a = new-object 'int[]' 10  
$i = 20 # out-of-bounds subscript

while ($true)  
{  
 try  
 {  
 $a[$i] = 10  
 "Assignment completed without error"  
 break  
 }

catch [IndexOutOfRangeException]  
 {  
 "Handling out-of-bounds index, >$\_<`n"  
 $i = 5  
 }

catch  
 {  
 "Caught unexpected exception"  
 }

finally  
 {  
 # …  
 }  
}

Windows PowerShell: Each exception thrown is raised as a System.Management.Automation.RuntimeException. If there are type-specific catch -clauses in the try block, the InnerException property of the exception is inspected to try and find a match, such as with the type System.IndexOutOfRangeException above.

## The trap statement

Syntax:

trap-statement:  
trap new-linesopt type-literalopt new-linesopt statement-block

Description:

A trap statement with and without type-literal is analogous to a catch block (§8.7) with and without catch-type-list, respectively, except that a trap statement can trap only one type at a time.

Multiple trap statements can be defined in the same statement-block, and their order of definition is irrelevant. If two trap statements with the same type-literal are defined in the same scope, the lexically first one is used to process an exception of matching type.

Unlike a catch block, a trap statement matches an exception type exactly; no derived type matching is performed.

When an exception occurs, if no matching trap statement is present in the current scope, a matching trap statement is searched for in the enclosing scope, which may involve looking in the calling script, function, or filter, and then in its caller, and so on. If the lookup ends up terminating all scopes, indicating that no handler exists for the exception, then the behavior is unspecified.

A trap statement's statement-body only executes to process the corresponding exception; otherwise, execution passes over it.

If a trap's statement-body exits normally, by default, an error object is written to the error stream, the exception is considered handled, and execution continues with the statement immediately following the one in the scope containing the trap statement that made the exception visible. [Note: The cause of the exception might be in a command called by the command containing the trap statement. end note]

If the final statement executed in a trap's statement-body is continue (§8.5.2), the writing of the error object to the error stream is suppressed, and execution continues with the statement immediately following the one in the scope containing the trap statement that made the exception visible. If the final statement executed in a trap's statement-body is break (§8.5.1), the writing of the error object to the error stream is suppressed, and the exception is re-thrown.

Within a trap statement the variable $\_ contains a description of the current error.

Consider the case in which an exception raised from within a try block does not have a matching catch block, but a matching trap statement exists at a higher block level. After the try block's finally clause is executed, the trap statement gets control even if any parent scope has a matching catch block. If a trap statement is defined within the try block itself, and that try block has a matching catch block, the trap statement gets control.

Examples:

In the following example, the error object is written and execution continues with the statement immediately following the one that caused the trap; that is, "Done" is written to the pipeline.

$j = 0; $v = 10/$j; "Done"  
trap { $j = 2 }

In the following example, the writing of the error object is suppressed and execution continues with the statement immediately following the one that caused the trap; that is, "Done" is written to the pipeline.

$j = 0; $v = 10/$j; "Done"  
trap { $j = 2; continue }

In the following example, the writing of the error object is suppressed and the exception is re-thrown.

$j = 0; $v = 10/$j; "Done"  
trap { $j = 2; break }

In the following example, the trap and exception-generating statements are in the same scope. After the exception is caught and handled, execution resumes with writing 1 to the pipeline.

&{trap{}; throw '...'; 1}

In the following example, the trap and exception-generating statements are in different scopes. After the exception is caught and handled, execution resumes with writing 2 (not 1) to the pipeline.

trap{} &{throw '...'; 1}; 2

## The data statement

Syntax:

data-statement:  
data new-linesopt data-name data-commands-allowedopt statement-block

data-name:  
simple-name

data-commands-allowed:  
new-linesopt -supportedcommand data-commands-list

data-commands-list:  
new-linesopt data-command  
data-commands-list , new-linesopt data-command

data-command:  
command-name-expr

Description:

A data statement creates a data section, keeping that section's data separate from the code. This separation supports facilities like separate string resource files for text, such as error messages and Help strings. It also helps support internationalization by making it easier to isolate, locate, and process strings that will be translated into different languages.

A script or function can have zero or more data sections.

The statement-block of a data section is limited to containing the following PowerShell features only:

* All operators except -match
* The if statement
* The following automatic variables: $PsCulture, $PsUICulture, $true, $false, and $null.
* Comments
* Pipelines
* Statements separated by semicolons (;)
* Literals
* Calls to the ConvertFrom-StringData cmdlet (§13.7)
* Any other cmdlets identified via the supportedcommand parameter

If the ConvertFrom-StringData cmdlet is used, the key/value pairs can be expressed using any form of string literal. However, expandable-string-literals and expandable-here-string-literals must not contain any variable substitutions or sub-expression expansions.

Examples:

The SupportedCommand parameter indicates that the given cmdlets or functions generate data only. For example, the following data section includes a user-written cmdlet, ConvertTo-XML, which formats data in an XML file:

data -supportedCommand ConvertTo-XML  
{  
 Format-XML -strings string1, string2, string3  
}

Consider the following example, in which the data section contains a ConvertFrom-StringData command that converts the strings into a hash table, whose value is assigned to $messages.

$messages = data  
{  
 ConvertFrom-StringData -stringdata @'  
 Greeting = Hello  
 Yes = yes  
 No = no  
'@  
}

The keys and values of the hash table are accessed using $messages.Greeting, $messages.Yes, and $messages.No, respectively.

Now, this can be saved as an English-language resource. German- and Spanish-language resources can be created in separate files, with the following data sections:

$messages = data  
{  
 ConvertFrom-StringData -stringdata @"  
 Greeting = Guten Tag  
 Yes = ja  
 No = nein  
"@  
}

$messagesS = data  
{  
 ConvertFrom-StringData -stringdata @"  
 Greeting = Buenos días  
 Yes = sí  
 No = no  
"@  
}

If dataname is present, it names the variable (without using a leading $) into which the value of the data statement is to be stored. Specifically, $name = data { … } is equivalent to data name { … }.

## Function definitions

Syntax:

function-statement:  
function new-linesopt function-name function-parameter-declarationopt { script-block }  
filter new-linesopt function-name function-parameter-declarationopt { script-block }

function-name:  
command-argument

command-argument:  
command-name-expr

function-parameter-declaration:  
new-linesopt ( parameter-list new-linesopt )

parameter-list:  
script-parameter  
parameter-list new-linesopt , script-parameter

script-parameter:  
new-linesopt attribute-listopt new-linesopt variable script-parameter-defaultopt

script-block:  
param-blockopt statement-terminatorsopt script-block-bodyopt

param-block:  
new-linesopt attribute-listopt new-linesopt param new-linesopt  
 ( parameter-listopt new-linesopt )

parameter-list:  
script-parameter  
parameter-list new-linesopt , script-parameter

script-parameter-default:  
new-linesopt = expression

script-block-body:  
named-block-list  
statement-list

named-block-list:  
named-block  
named-block-list named-block

named-block:  
block-name statement-block statement-terminatorsopt

block-name: one of  
dynamicparam begin process end

Description:

A function definition specifies the name of the function or filter being defined and the names of its parameters, if any. It also contains zero or more statements that are executed to achieve that function's purpose.

Windows PowerShell: Each function is an instance of the class System.Management.Automation.FunctionInfo.

### Filter functions

Whereas an ordinary function runs once in a pipeline and accesses the input collection via $input, a filter is a special kind of function that executes once for each object in the input collection. The object currently being processed is available via the variable $\_.

A filter with no named blocks (§8.10.6) is equivalent to a function with a process block, but without any begin block or end block.

Consider the following filter function definition and calls:

filter Get-Square2 # make the function a filter   
{  
 $\_ \* $\_ # access current object from the collection  
}  
  
-3..3 | Get-Square2 # collection has 7 elements  
6,10,-3 | Get-Square2 # collection has 3 elements

Windows PowerShell: Each filter is an instance of the class System.Management.Automation.FilterInfo (§4.5.11).

### Argument processing

Consider the following definition for a function called Get-Power:

function Get-Power ([long]$base, [int]$exponent)  
{  
 $result = 1  
 for ($i = 1; $i -le $exponent; ++$i)   
 {  
 $result \*= $base  
 }  
 return $result  
}

This function has two parameters, $base and $exponent. It also contains a set of statements that, for non-negative exponent values, computes $base$exponent and returns the result to Get-Power's caller.

When a script, function, or filter begins execution, each parameter is initialized to its corresponding argument's value. If there is no corresponding argument and a default value (§8.10.3) is supplied, that value is used; otherwise, the value $null is used. As such, each parameter is a new variable just as if it was initialized by assignment at the start of the script-block.

If a script-parameter contains a type constraint (such as [long] and [int] above), the value of the corresponding argument is converted to that type, if necessary; otherwise, no conversion occurs.

When a script, function, or filter begins execution, variable $args is defined inside it as an unconstrained 1-dimensional array, which contains all arguments not bound by name or position, in lexical order.

Consider the following function definition and calls:

function F ($a, $b, $c, $d) { … }  
  
F -b 3 -d 5 2 4 # $a is 2, $b is 3, $c is 4, $d is 5, $args Length 0  
F -a 2 -d 3 4 5 # $a is 2, $b is 4, $c is 5, $d is 3, $args Length 0  
F 2 3 4 5 -c 7 -a 1 # $a is 1, $b is 2, $c is 7, $d is 3, $args Length 2

For more information about parameter binding see §8.11.

### Parameter initializers

The declaration of a parameter p may contain an initializer, in which case, that initializer's value is used to initialize p provided p is not bound to any arguments in the call.

Consider the following function definition and calls:

function Find-Str ([string]$str, [int]$start\_pos = 0) { … }  
  
Find-Str "abcabc" # 2nd argument omitted, 0 used for $start\_pos  
Find-Str "abcabc" 2 # 2nd argument present, so it is used for $start\_pos

### The [switch] type constraint

When a switch parameter is passed, the corresponding parameter in the command must be constrained by the type switch. Type switch has two values, True and False.

Consider the following function definition and calls:

function Process ([switch]$trace, $p1, $p2) { … }  
  
Process 10 20 # $trace is False, $p1 is 10, $p2 is 20  
Process 10 -trace 20 # $trace is True, $p1 is 10, $p2 is 20  
Process 10 20 -trace # $trace is True, $p1 is 10, $p2 is 20  
Process 10 20 -trace:$false # $trace is False, $p1 is 10, $p2 is 20  
Process 10 20 -trace:$true # $trace is True, $p1 is 10, $p2 is 20

### Pipelines and functions

When a script, function, or filter is used in a pipeline, a collection of values is delivered to that script or function. The script, function, or filter gets access to that collection via the enumerator $input (§2.3.2.2, §4.5.16), which is defined on entry to that script, function, or filter.

Consider the following function definition and calls:

function Get-Square1  
{  
 foreach ($i in $input) # iterate over the collection   
 {  
 $i \* $i  
 }  
}  
  
-3..3 | Get-Square1 # collection has 7 elements  
6,10,-3 | Get-Square1 # collection has 3 elements

### Named blocks

The statements within a script-block can belong to one large unnamed block, or they can be distributed into one or more named blocks. Named blocks allow custom processing of collections coming from pipelines; named blocks can be defined in any order.

The statements in a begin block (i.e.; one marked with the keyword begin) are executed once, before the first pipeline object is delivered.

The statements in a process block (i.e.; one marked with the keyword process) are executed for each pipeline object delivered. ($\_ provides access to the current object being processed from the input collection coming from the pipeline.) This means that if a collection of zero elements is sent via the pipeline, the process block is not executed at all. However, if the script or function is called outside a pipeline context, this block is executed exactly once, and $\_ is set to $null, as there is no current collection object.

The statements in an end block (i.e.; one marked with the keyword end) are executed once, after the last pipeline object has been delivered.

### dynamicParam block

The subsections of §8.10 thus far deal with static parameters, which are defined as part of the source code. It is also possible to define dynamic parameters via a dynamicParam block, another form of named block (§8.10.6), which is marked with the keyword dynamicParam. Much of this machinery is implementation defined.

Dynamic parameters are parameters of a cmdlet, function, filter, or script that are available under certain conditions only. One such case is the Encoding parameter of the Set-Item cmdlet.

In the statement-block, use an if statement to specify the conditions under which the parameter is available in the function. Use the New-Object cmdlet (§13.35) to create an object of an implementation-defined type to represent the parameter, and specify its name. Also, use New-Object to create an object of a different implementation-defined type to represent the implementation-defined attributes of the parameter.

Windows PowerShell: The following example shows a function with standard parameters called Name and Path, and an optional dynamic parameter named DP1. The DP1 parameter is in the PSet1 parameter set and has a type of Int32. The DP1 parameter is available in the Sample function only when the value of the Path parameter contains "HKLM:", indicating that it is being used in the HKEY\_LOCAL\_MACHINE registry drive.

function Sample  
{  
 Param ([String]$Name, [String]$Path)  
 DynamicParam  
 {  
 if ($path -match "\*HKLM\*:")  
 {  
 $dynParam1 = New-Object"  
 System.Management.Automation.RuntimeDefinedParameter("dp1",  
 [Int32], $attributeCollection)

$attributes = New-Object  
 System.Management.Automation.ParameterAttribute  
 $attributes.ParameterSetName = 'pset1'  
 $attributes.Mandatory = $false

$attributeCollection = New-Object -Type  
 System.Collections.ObjectModel.Collection``1[System.Attribute]  
 $attributeCollection.Add($attributes)

$paramDictionary = New-Object   
 System.Management.Automation.RuntimeDefinedParameterDictionary  
 $paramDictionary.Add("dp1", $dynParam1)  
 return $paramDictionary  
 }  
 }  
}

Windows PowerShell: The type used to create an object to represent a dynamic parameter is System.Management.Automation.RuntimeDefinedParameter.

Windows PowerShell: The type used to create an object to represent the attributes of the parameter is System.Management.Automation.ParameterAttribute.

Windows PowerShell: The implementation-defined attributes of the parameter include Mandatory, Position, and ValueFromPipeline.

### param block

A param-block provides an alternate way of declaring parameters. For example, the following sets of parameter declarations are equivalent:

function FindStr1 ([string]$str, [int]$start\_pos = 0) { … }  
function FindStr2 { param ([string]$str, [int]$start\_pos = 0) … }

A param-block allows an attribute-list on the param-block whereas a function-parameter-declaration does not.

A script may have a param-block but not a function-parameter-declaration. A function or filter definition may have a function-parameter-declaration or a param-block, but not both.

Consider the following example:

param ( [Parameter(Mandatory = $true, ValueFromPipeline=$true)]  
 [string[]] $ComputerName )

The one parameter, $ComputerName, has type string[], it is required, and it takes input from the pipeline.

See §12.3.7 for a discussion of the Parameter attribute and for more examples.

## Parameter binding

When a script, function, filter, or cmdlet is invoked, each argument can be bound to the corresponding parameter by position, with the first parameter having position zero.

Consider the following definition fragment for a function called Get-Power, and the calls to it:

function Get-Power ([long]$base, [int]$exponent) { … }

Get-Power 5 3 # argument 5 is bound to parameter $base in position 0  
 # argument 3 is bound to parameter $exponent in position 1  
 # no conversion is needed, and the result is 5 to the power 3

Get-Power 4.7 3.2 # double argument 4.7 is rounded to int 5, double argument  
 # 3.2 is rounded to int 3, and result is 5 to the power 3

Get-Power 5 # $exponent has value $null, which is converted to int 0

Get-Power # both parameters have value $null, which is converted to int 0

When a script, function, filter, or cmdlet is invoked, an argument can be bound to the corresponding parameter by name. This is done by using a parameter with argument, which is an argument that is the parameter's name with a leading dash (-), followed by the associated value for that argument. The parameter name used can have any case-insensitive spelling and can use any prefix that uniquely designates the corresponding parameter. [Note: When choosing parameter names, avoid using the names of the common parameters (§13.56). end note]

Consider the following calls to function Get-Power:

Get-Power -base 5 -exponent 3 # -base designates $base, so 5 is   
 # bound to that, exponent designates  
 # $exponent, so 3 is bound to that

Get-Power -Exp 3 -BAs 5 # $base takes on 5 and $exponent takes on 3

Get-Power -e 3 -b 5 # $base takes on 5 and $exponent takes on 3

On the other hand, calls to the following function

function Get-Hypot ([double]$side1, [double]$side2)  
{  
 return [Math]::Sqrt($side1 \* $side1 + $side2 \* $side2)  
}

must use parameters -side1 and -side2, as there is no prefix that uniquely designates the parameter.

The same parameter name cannot be used multiple times with or without different associated argument values.

Parameters can have attributes (§12). For information about the individual attributes see the sections within §12.3. For information about parameter sets see §12.3.7.

A script, function, filter, or cmdlet can receive arguments via the invocation command line, from the pipeline, or from both. Here are the steps, in order, for resolving parameter binding:

1. Bind all named parameters, then
2. Bind positional parameters, then
3. Bind from the pipeline by value (§12.3.7) with exact match, then
4. Bind from the pipeline by value (§12.3.7) with conversion, then
5. Bind from the pipeline by name (§12.3.7) with exact match, then
6. Bind from the pipeline by name (§12.3.7) with conversion

Several of these steps involve conversion, as described in §6. However, the set of conversions used in binding is not exactly the same as that used in language conversions. Specifically,

* Although the value $null can be cast to bool, $null cannot be bound to bool.
* When the value $null is passed to a switch parameter for a cmdlet, it is treated as if $true was passed. However, when passed to a switch parameter for a function, it is treated as if $false was passed.
* Parameters of type bool or switch can only bind to numeric or bool arguments.
* If the parameter type is not a collection, but the argument is some sort of collection, no conversion is attempted unless the parameter type is object or PsObject.  (The main point of this restriction is to disallow converting a collection to a string parameter.) Otherwise, the usual conversions are attempted.

Windows PowerShell: If the parameter type is IList or ICollection<T>, only those conversions via Constructor, op\_Implicit, and op\_Explicit are attempted. If no such conversions exist, a special conversion for parameters of “collection” type is used, which includes IList, ICollection<T>, and arrays.

Positional parameters prefer to be bound without type conversion, if possible. For example,

function Test  
{  
 [CmdletBinding(DefaultParameterSetname = "SetB")]  
 param([Parameter(Position = 0, ParameterSetname = "SetA")]  
 [decimal]$dec,  
 [Parameter(Position = 0, ParameterSetname = "SetB")]  
 [int]$in  
 )  
 $PsCmdlet.ParameterSetName  
}  
  
Test 42d # outputs "SetA"  
Test 42 # outputs "SetB"

# Arrays

## Introduction

PowerShell supports arrays of one or more dimensions with each dimension having zero or more elements. Within a dimension, elements are numbered in ascending integer order starting at zero. Any individual element can be accessed via the array subscript operator [] (§7.1.4). The number of dimensions in an array is called its rank.

An element can contain a value of any type including an array type. An array having one or more elements whose values are of any array type is called a jagged array. A multidimensional array has multiple dimensions, in which case, the number of elements in each row of a dimension is the same. An element of a jagged array may contain a multidimensional array, and vice versa.

Multidimensional arrays are stored in row-major order. The number of elements in an array is called that array's length, which is fixed when the array is created. As such, the elements in a 1-dimensional array A having length N can be accessed (i.e., subscripted) using the expressions A[0], A[1], …, A[N-1]. The elements in a 2-dimensional array B having M rows, with each row having N columns, can be accessed using the expressions B[0,0], B[0,1], …, B[0,N-1], B[1,0], B[1,1], …, B[1,N-1], …, B[M-1,0], B[M-1,1], …, B[M-1,N-1]. And so on for arrays with three or more dimensions.

By default, an array is polymorphic; i.e., its elements need not all have the same type. For example,

$items = 10,"blue",12.54e3,16.30D # 1-D array of length 4  
$items[1] = -2.345  
$items[2] = "green"

$a = New-Object 'object[,]' 2,2 # 2-D array of length 4  
$a[0,0] = 10  
$a[0,1] = $false  
$a[1,0] = "red"  
$a[1,1] = $null

A 1-dimensional array has type type[], a 2-dimensional array has type type[,],a 3-dimensional array has type type[,,], and so on, where type is object for an unconstrained type array, or the constrained type for a constrained array (§9.4).

All array types are derived from the type Array (§4.3.2).

## Array creation

An array is created via an array creation expression, which has the following forms: unary comma operator (§7.2.1) ,array-expression (§7.1.7), binary comma operator (§7.3), range operator (§7.4), or New-Object cmdlet (§13.36).

Here are some examples of array creation and usage:

$values = 10,20,30  
for ($i = 0; $i -lt $values.Length; ++$i)  
{  
 "`$values[$i] = $($values[$i])"  
}

$x = ,10 # x refers to an array of length 1  
$x = @(10) # x refers to an array of length 1  
$x = @() # x refers to an array of length 0

$a = New-Object 'object[,]' 2,2 # create a 2x2 array of anything  
$a[0,0] = 10 # set to an int value  
$a[0,1] = $false # set to a boolean value  
$a[1,0] = "red" # set to a string value  
$a[1,1] = 10.50D # set to a decimal value  
foreach ($e in $a) # enumerate over the whole array  
{  
 $e  
}

The following is written to the pipeline:

$values[0] = 10  
$values[1] = 20  
$values[2] = 30

10  
False  
red  
10.50

The default initial value of any element not explicitly initialized is the default value for that element's type (that is, $false, zero, or $null).

## Array concatenation

Arrays of arbitrary type and length can be concatenated via the + and += operators, both of which result in the creation of a new unconstrained 1-dimensional array. The existing arrays are unchanged. See §7.7.3 for more information, and §9.4 for a discussion of adding to an array of constrained type.

## Constraining element types

A 1-dimensional array can be created so that it is type-constrained by prefixing the array-creation expression with an array type cast. For example,

$a = [int[]](1,2,3,4) # constrained to int  
$a[1] = "abc" # implementation-defined behavior  
$a += 1.23 # new array is unconstrained

The syntax for creating a multidimensional array requires the specification of a type (§13.36), and that type becomes the constraint type for that array. However, by specifying type object[], there really is no constraint as a value of any type can be assigned to an element of an array of that type.

Concatenating two arrays (§7.7.3) always results in a new array that is unconstrained even if both arrays are constrained by the same type. For example,

$a = [int[]](1,2,3) # constrained to int  
$b = [int[]](10,20) # constrained to int  
$c = $a + $b # constraint not preserved  
$c = [int[]]($a + $b) # result explicitly constrained to int

## Arrays as reference types

As array types are reference types, a variable designating an array can be made to refer to any array of any rank, length, and element type. For example,

$a = 10,20 # $a refers to an array of length 2  
$a = 10,20,30 # $a refers to a different array, of length 3  
$a = "red",10.6 # $a refers to a different array, of length 2  
$a = New-Object 'int[,]' 2,3 # $a refers to an array of rank 2

Assignment of an array involves a shallow copy; that is, the variable assigned to refers to the same array, no copy of the array is made. For example,

$a = 10,20,30  
">$a<"  
$b = $a # make $b refer to the same array as $a  
">$b<"

$a[0] = 6 # change value of [0] via $a  
">$a<"  
">$b<" # change is reflected in $b

$b += 40 # make $b refer to a new array  
$a[0] = 8 # change value of [0] via $a  
">$a<"  
">$b<" # change is not reflected in $b

The following is written to the pipeline:

>10 20 30<  
>10 20 30<  
>6 20 30<  
>6 20 30<  
>8 20 30<  
>6 20 30 40<

## Arrays as array elements

Any element of an array can itself be an array. For example,

$colors = "red", "blue", "green"  
$list = $colors, (,7), (1.2, "yes") # parens in (,7) are redundant; they  
 # are intended to aid readability  
"`$list refers to an array of length $($list.Length)"  
">$($list[1][0])<"  
">$($list[2][1])<"

The following is written to the pipeline:

$list refers to an array of length 3  
>7<  
>yes<

$list[1] refers to an array of 1 element, the integer 7, which is accessed via $list[1][0], as shown. Compare this with the following subtly different case:

$list = $colors, 7, (1.2, "yes") # 7 has no prefix comma  
">$($list[1])<"

Here, $list[1] refers to a scalar, the integer 7, which is accessed via $list[1].

Consider the following example,

$x = [string[]]("red","green")  
$y = 12.5, $true, "blue"  
$a = New-Object 'object[,]' 2,2  
$a[0,0] = $x # element is an array of 2 strings  
$a[0,1] = 20 # element is an int  
$a[1,0] = $y # element is an array of 3 objects  
$a[1,1] = [int[]](92,93) # element is an array of 2 ints

## Negative subscripting

This is discussed in §7.1.4.1.

## Bounds checking

This is discussed in §7.1.4.1.

## Array slices

An array slice is an unconstrained 1-dimensional array whose elements are copies of zero or more elements from a collection. An array slice is created via the subscript operator [] (§7.1.4.5).

## Copying an array

A contiguous set of elements can be copied from one array to another using the method [Array]::Copy. For example,

$a = [int[]](10,20,30)  
$b = [int[]](0,1,2,3,4,5)  
[Array]::Copy($a, $b, 2) # $a[0]->$b[0], $a[1]->$b[1]  
[Array]::Copy($a, 1, $b, 3, 2) # $a[1]->$b[3], $a[2]->$b[4]

## Enumerating over an array

Although it is possible to loop through an array accessing each of its elements via the subscript operator, we can enumerate over that array's elements using the foreach statement. For a multidimensional array, the elements are processed in row-major order. For example,

$a = 10,53,16,-43  
foreach ($elem in $a) {  
 # do something with element via $e  
}

foreach ($elem in -5..5) {  
 # do something with element via $e  
}

$a = New-Object 'int[,]' 3,2  
foreach ($elem in $a) {  
 # do something with element via $e  
}

## Multidimensional array flattening

Some operations on a multidimensional array—such as replication (§7.6.3) and concatenation (§7.7.3)—require that array to be flattened; that is, to be turned into a 1-dimensional array of unconstrained type. The resulting array takes on all the elements in row-major order.

Consider the following example:

$a = "red",$true  
$b = (New-Object 'int[,]' 2,2)  
$b[0,0] = 10  
$b[0,1] = 20  
$b[1,0] = 30  
$b[1,1] = 40  
$c = $a + $b

The array designated by $c contains the elements "red", $true, 10, 20, 30, and 40.

# Hashtables

Syntax:

hash-literal-expression:  
@{ new-linesopt hash-literal-bodyopt new-linesopt }

hash-literal-body:  
hash-entry  
hash-literal-body statement-terminators hash-entry

hash-entry:  
key-expression = new-linesopt statement

key-expression:  
simple-name  
unary-expression

statement-terminator:  
;  
new-line-character

## Introduction

The type Hashtable represents a collection of key/value pair objects that supports efficient retrieval of a value when indexed by the key. Each key/value pair is an element, which is stored in some implementation-defined object type.

An element's key cannot be the null value. There are no restrictions on the type of a key or value. Duplicate keys are not supported.

Given a key/value pair object, the key and associated value can be obtained by using the instance properties Key and Value, respectively.

Given one or more keys, the corresponding value(s) can be accessed via the Hashtable subscript operator [] (§7.1.4.3).

All Hashtables have type Hashtable (§4.3.3).

The order of the keys in the collection returned by Keys is unspecified; however, it is the same order as the associated values in the collection returned by Values.

Here are some examples involving Hashtables:

$h1 = @{ FirstName = "James"; LastName = "Anderson"; IDNum = 123 }  
$h1.FirstName # designates the key FirstName  
$h1["LastName"] # designates the associated value for key LastName  
$h1.Keys # gets the collection of keys

Windows PowerShell: Hashtable elements are stored in an object of type DictionaryEntry, and the collections returned by Keys and Values have type ICollection.

## Hashtable creation

A Hashtable is created via a hash literal (§7.1.9) or the New-Object cmdlet (§13.36). It can be created with zero or more elements. The Count property returns the current element count.

## Adding and removing Hashtable elements

An element can be added to a Hashtable by assigning (§7.11.1) a value to a non-existent key name or to a subscript (§7.1.4.3) that uses a non-existent key name. Removal of an element requires the use of the Remove method. For example,

$h1 = @{ FirstName = "James"; LastName = "Anderson"; IDNum = 123 } $h1.Dept = "Finance" # adds element Finance  
$h1["Salaried"] = $false # adds element Salaried  
$h1.Remove("Salaried") # removes element Salaried

## Hashtable concatenation

Hashtables can be concatenated via the + and += operators, both of which result in the creation of a new Hashtable. The existing Hashtables are unchanged. See §7.7.4 for more information.

## Hashtables as reference types

As Hashtable is a reference type, assignment of a Hashtable involves a shallow copy; that is, the variable assigned to refers to the same Hashtable; no copy of the Hashtable is made. For example,

$h1 = @{ FirstName = "James"; LastName = "Anderson"; IDNum = 123 }  
$h2 = $h1  
$h1.FirstName = "John" # change key's value in $h1  
$h2.FirstName # change is reflected in $h2

## Enumerating over a Hashtable

To process every pair in a Hashtable, use the Keys property to retrieve the list of keys as an array, and then enumerate over the elements of that array getting the associated value via the Value property or a subscript, as follows

$h1 = @{ FirstName = "James"; LastName = "Anderson"; IDNum = 123 }  
foreach ($e in $h1.Keys)  
{  
 "Key is " + $e + ", Value is " + $h1[$e]  
}

# Modules

## Introduction

As stated in §3.14, a module is a self-contained reusable unit that allows PowerShell code to be partitioned, organized, and abstracted. A module can contain one or more module members, which are commands (such as cmdlets and functions) and items (such as variables and aliases). The names of these members can be kept private to the module or they may be exported to the session into which the module is imported.

There are three different module types: manifest, script, and binary. A manifest module is a file that contains information about a module, and controls certain aspects of that module's use. A script module is a PowerShell script file with a file extension of ".psm1" instead of ".ps1". A binary module contains class types that define cmdlets and providers. Unlike script modules, binary modules are written in compiled languages. Binary modules are not covered by this specification.

Windows PowerShell: A binary module is a .NET assembly (i.e.; a DLL) that was compiled against the PowerShell libraries.

Modules may nest; that is, one module may import another module. A module that has associated nested modules is a root module.

When a PowerShell session is created, by default, no modules are imported.

When modules are imported, the search path used to locate them is defined by the environment variable PSModulePath.

The following cmdlets deal with modules:

* Get-Module: Identifies the modules that have been, or that can be, imported (see §13.23)
* Import-Module: Adds one or more modules to the current session (see §11.4, §13.28)
* Export-ModuleMember: Identifies the module members that are to be exported (see §13.11)
* Remove-Module: Removes one or more modules from the current session (see §11.5, §13.41)
* New-Module: Creates a dynamic module (see §11.7, §13.35)

## Writing a script module

A script module is a script file. Consider the following script module:

function Convert-CentigradeToFahrenheit ([double]$tempC)  
{  
 return ($tempC \* (9.0/5.0)) + 32.0  
}  
New-Alias c2f Convert-CentigradeToFahrenheit

function Convert-FahrenheitToCentigrade ([double]$tempF)  
{  
 return ($tempF - 32.0) \* (5.0/9.0)  
}  
New-Alias f2c Convert-FahrenheitToCentigrade

Export-ModuleMember -Function Convert-CentigradeToFahrenheit  
Export-ModuleMember -Function Convert-FahrenheitToCentigrade  
Export-ModuleMember -Alias c2f,f2c

This module contains two functions, each of which has an alias. By default, all function names, and only function names are exported. However, once the cmdlet Export-ModuleMember has been used to export anything, then only those things exported explicitly will be exported. A series of commands and items can be exported in one call or a number of calls to this cmdlet; such calls are cumulative for the current session.

## Installing a script module

A script module is defined in a script file, and modules can be stored in any directory. The environment variable PSModulePath points to a set of directories to be searched when module-related cmdlets look for modules whose names do not include a fully qualified path. Additional lookup paths can be provided; for example,

$Env:PSModulepath = $Env:PSModulepath + ";<additional-path>"

Any additional paths added affect the current session only.

Alternatively, a fully qualified path can be specified when a module is imported (§13.28).

## Importing a script module

Before the resources in a module can be used, that module must be imported into the current session, using the cmdlet Import-Module (§13.28). Import-Module can restrict the resources that it actually imports.

When a module is imported, its script file is executed. That process can be configured by defining one or more parameters in the script file, and passing in corresponding arguments via the ArgumentList parameter of Import-Module.

Consider the following script that uses these functions and aliases defined in §11.2:

Import-Module "E:\Scripts\Modules\PSTest\_Temperature" -Verbose  
  
"0 degrees C is " + (Convert-CentigradeToFahrenheit 0) + " degrees F"  
"100 degrees C is " + (c2f 100) + " degrees F"  
"32 degrees F is " + (Convert-FahrenheitToCentigrade 32) + " degrees C"  
"212 degrees F is " + (f2c 212) + " degrees C"

Importing a module causes a name conflict when commands or items in the module have the same names as commands or items in the session. A name conflict results in a name being hidden or replaced. The Prefix parameter of Import-Module can be used to avoid naming conflicts. Also, the Alias, Cmdlet, Function, and Variable parameters can limit the selection of commands to be imported, thereby reducing the chances of name conflict.

Even if a command is hidden, it can be run by qualifying its name with the name of the module in which it originated. For example, & M\F 100 invokes the function F in module M, and passes it the argument 100.

When the session includes commands of the same kind with the same name, such as two cmdlets with the same name, by default it runs the most recently added command.

See §3.5.6 for a discussion of scope as it relates to modules.

## Removing a script module

One or more modules can be removed from a session via the cmdlet Remove-Module (§13.41).

Removing a module does not uninstall the module.

Windows PowerShell: In a script module, it is possible to specify code that is to be executed prior to that module's removal, as follows:

$MyInvocation.MyCommand.ScriptBlock.Module.OnRemove = { on-removal-code }

## Module manifests

As stated in §11.1, a manifest module is a file that contains information about a module, and controls certain aspects of that module's use.

A module need not have a corresponding manifest, but if it does, that manifest has the same name as the module it describes, but with a .psd1 file extension.

A manifest contains a limited subset of PowerShell script, which returns a Hashtable containing a set of keys. These keys and their values specify the manifest elements for that module. That is, they describe the contents and attributes of the module, define any prerequisites, and determine how the components are processed.

Essentially, a manifest is a data file; however, it can contain references to data types, the if statement, and the arithmetic and comparison operators. (Assignments, function definitions and loops are not permitted.) A manifest also has read access to environment variables and it can contain calls to the cmdlet Join-Path, so paths can be constructed.

Here are the keys permitted in a manifest:

| **Key** | **Value** | **Description** |
| --- | --- | --- |
| AliasesToExport | string[] | Default value: \* (all aliases that are exported)  Specifies the aliases that the module exports. Wildcards are permitted. While it can remove aliases from the list of exported aliases, it cannot add aliases to that list. |
| Author | string | Default value: "<username>"  The module author name. |
| ClrVersion | string | Windows PowerShell: Default value: ""  Specifies the minimum version of the Common Language Runtime (CLR) of the Microsoft .NET Framework that the module requires. |
| CmdletsToExport | string[] | Default value: \* (all cmdlets that are exported)  Specifies the cmdlets that the module exports. Wildcards are permitted. While it can remove cmdlets from the list of exported cmdlets, it cannot add cmdlets to that list. |
| CompanyName | string | Default value: "Unknown"  Identifies the company or vendor that created the module. |
| Copyright | string | Default value: "(c) <year> <username>. All rights reserved." where <year> is the current year and <username> is the value of the Author key (if one is specified) or the name of the current user.  Specifies a copyright statement for the module. |
| Description | string | Default value: ""  Description of the functionality provided by this module. |
| DotNetFrameworkVersion | string | Windows PowerShell: Default value: ""  Specifies the minimum version of the Microsoft .NET Framework that the module requires. |
| FileList | string[] | Default value: ""  Specifies all items that are packaged with the module. This key is designed to act as a module inventory. These files are not automatically exported with the module. |
| FormatsToProcess | string[] | Windows PowerShell: Default value: @()  Specifies the formatting files (.ps1xml) that run when the module is imported.  When a module is imported, PowerShell runs the Update-FormatData cmdlet with the specified files. As formatting files are not scoped, they affect the whole session. |
| FunctionsToExport | string[] | Default value: \* (all functions are exported)  Specifies the functions that the module exports. Wildcards are permitted. While it can remove functions from the list of exported functions, it cannot add functions to that list. |
| ModuleList | string[] | Default value: @()  Lists all modules that are packaged with this module. This key is designed to act as a module inventory. These modules are not automatically processed. |
| ModuleToProcess | string | Default value: ""  Specifies the primary or root file of the module. When the module is imported, the members that are exported from the root module file are imported into the caller's session state. Enter the file name of a script module or binary module.  If a module has a manifest file and no root file has been designated in the ModuleToProcess key, the manifest becomes the primary file for the module, and the module becomes a manifest module (i.e., it's (ModuleType is Manifest).  To export members from a module that has a manifest, the names of those files must be specified in the values of the ModuleToProcess or NestedModules keys in the manifest. Otherwise, their members are not exported. |
| ModuleVersion | string | Default value: "1.0".  Specifies the version of the module. |
| NestedModules | string[] | Default value: @()  Specifies script modules and binary modules that are imported into the module's session state. The files in the NestedModules key run in the order in which they are listed in the value.  Typically, nested modules contain commands that the root module needs for its internal processing. By default, the commands in nested modules are exported from the module's session state into the caller's session state, but the root module can restrict the commands that it exports (for example, by using an Export-Module command).  Nested modules in the module session state are available to the root module, but they are not returned by a Get-Module command in the caller's session state.  Scripts that are listed in the NestedModules key are run in the module's session state, not in the caller's session state. To run a script in the caller's session state, list the script file name in the value of the ScriptsToProcess key in the manifest. |
| PowerShellHostName | string | Default value: ""  Specifies the name of the PowerShell host program that the module requires. |
| PowerShellHostVersion | string | Default value: ""  Specifies the minimum version of the PowerShell host program that works with the module. |
| PowerShellVersion | string | Default value: ""  Specifies the minimum version of PowerShell that will work with this module. |
| PrivateData | string | Default value: ""  Specifies private data to pass to the module specified in ModuleToProcess. |
| ProcessorArchitecture | string | Default value: ""  Specifies the processor architecture that the module requires. Valid values are x86, AMD64, IA64, and None (unknown or unspecified). |
| RequiredAssemblies | string[] | Default value: @()  Specifies the assemblies that must be loaded prior to importing this module |
| RequiredModules | string[] | Default value: @()  Specifies the modules that must be imported into the global environment prior to importing this module |
| ScriptsToProcess | string[] | Default value: @()  Specifies script files that run in the caller's session state when the module is imported. These scripts can be used to prepare an environment. To specify scripts that run in the module's session state, use the NestedModules key. |
| TypesToProcess | string[] | Windows PowerShell: Default value: @()  Specifies the type files (.ps1xml) to be loaded when importing this module.  When a module is imported, PowerShell runs the Update-TypeData cmdlet with the specified files. As type files are not scoped, they affect all session states in the session. |
| VariablesToExport | string[] | Default value: \* (all variables are exported)  Specifies the variables that the module exports. Wildcards are permitted. While it can remove variables from the list of exported variables, it cannot add variables to that list. |

The only key that is required is ModuleVersion.

Here is an example of a simple manifest:

@{  
 ModuleVersion = '1.0'  
 Author = 'John Doe'  
 RequiredModules = @()  
 FunctionsToExport = 'Set\*','Get\*','Process\*'  
}

Windows PowerShell: The key GUID has a string Value. This specifies a Globally Unique IDentifier (GUID) for the module. The GUID can be used to distinguish among modules having the same name. To create a new GUID, call the method [guid]::NewGuid().

## Dynamic modules

A dynamic module is a module that is created in memory at runtime by the cmdlet New-Module (§13.35); it is not loaded from disk. Consider the following example:

$sb = {  
 function Convert-CentigradeToFahrenheit ([double]$tempC)  
 {  
 return ($tempC \* (9.0/5.0)) + 32.0  
 }

New-Alias c2f Convert-CentigradeToFahrenheit

function Convert-FahrenheitToCentigrade ([double]$tempF)  
 {  
 return ($tempF - 32.0) \* (5.0/9.0)  
 }

New-Alias f2c Convert-FahrenheitToCentigrade

Export-ModuleMember -Function Convert-CentigradeToFahrenheit  
 Export-ModuleMember -Function Convert-FahrenheitToCentigrade  
 Export-ModuleMember -Alias c2f,f2c  
}

New-Module -Name MyDynMod -ScriptBlock $sb  
Convert-CentigradeToFahrenheit 100  
c2f 100

The script block $sb defines the contents of the module, in this case, two functions and two aliases to those functions. As with an on-disk module, only functions are exported by default, so Export-ModuleMember cmdlets calls exist to export both the functions and the aliases.

Once New-Module runs, the four names exported are available for use in the session, as is shown by the calls to the Convert-CentigradeToFahrenheit and c2f.

Like all modules, the members of dynamic modules run in a private module scope that is a child of the global scope. Get-Module cannot get a dynamic module, but Get-Command can get the exported members.

To make a dynamic module available to Get-Module, pipe a New-Module command to Import-Module, or pipe the module object that New-Module returns, to Import-Module. This action adds the dynamic module to the Get-Module list, but it does not save the module to disk or make it persistent.

## Closures

A dynamic module can be used to create a closure, a function with attached data. Consider the following example:

function Get-NextID ([int]$startValue = 1)  
{  
 $nextID = $startValue  
 {  
 ($script:nextID++)  
 }.GetNewClosure()  
}

$v1 = Get-NextID # get a scriptblock with $startValue of 0  
&$v1 # invoke Get-NextID getting back 1  
&$v1 # invoke Get-NextID getting back 1

$v2 = Get-NextID 100 # get a scriptblock with $startValue of 100  
&$v2 # invoke Get-NextID getting back 100  
&$v2 # invoke Get-NextID getting back 101

The intent here is that Get-NextID return the next ID in a sequence whose start value can be specified. However, multiple sequences must be supported, each with its own $startValue and $nextID context. This is achieved by the call to the method [scriptblock]::GetNewClosure (§4.3.7).

Each time a new closure is created by GetNewClosure, a new dynamic module is created, and the variables in the caller‘s scope (in this case, the script block containing the increment) are copied into this new module. To ensure that the nextId defined inside the parent function (but outside the script block) is incremented, the explicit script: scope prefix is needed.

Of course, the script block need not be a named function; for example:

$v3 = &{ # get a scriptblock with $startValue of 200  
 param ([int]$startValue = 1)  
 $nextID = $startValue  
 {  
 ($script:nextID++)  
 }.GetNewClosure()  
} 200

&$v3 # invoke script getting back 200  
&$v3 # invoke script getting back 201

# Attributes

An attribute object associates predefined system information with a target element, which can be a param block or a parameter (§8.10). Each attribute object has an attribute type.

Information provided by an attribute is also known as metadata. Metadata can be examined by the command or the execution environment to control how the command processes data or before run time by external tools to control how the command itself is processed or maintained.

Multiple attributes can be applied to the same target element.

## Attribute specification

attribute-list:  
attribute  
attribute-list attribute

attribute:  
[ attribute-name ( attribute-arguments new-linesopt ) new-linesopt ]

attribute-name:  
type-spec

attribute-arguments:  
attribute-argument  
attribute-argument new-linesopt , attribute-arguments

attribute-argument:  
new-linesopt expression  
new-linesopt simple-name = new-linesopt expression

An attribute consists of an attribute-name and an optional list of positional and named arguments. The positional arguments (if any) precede the named arguments. A positional argument consists of a simple-name, followed by an equal sign, followed by an expression.

The attribute-name is a reserved attribute type (§12.3) or some implementation-defined attribute type.

## Attribute instances

An attribute instance is an object of an attribute type. The instance represents an attribute at run-time.

To create an object of some attribute type A, use the notation A(). An attribute is declared by enclosing its instance inside [ ], as in [A()]. Some attribute types have positional and named parameters (§8.11), just like functions and cmdlets. For example,

[A(10,IgnoreCase=$true)]

shows an instance of type A being created using a positional parameter whose argument value is 10, and a named parameter, IgnoreCase, whose argument value is $true.

## Reserved attributes

The attributes described in the following sections can be used to augment or modify the behavior of PowerShell functions, filters, scripts, and cmdlets.

### The Alias attribute

This attribute is used in a script-parameter to specify an alternate name for a parameter. A parameter may have multiple aliases, and each alias name must be unique within a parameter-list. One possible use is to have different names for a parameter in different parameter sets (see ParameterSetName).

The attribute argument has type string[].

Consider a function call Test1 that has the following param block, and which is called as shown:

param ( [Parameter(Mandatory = $true)]  
 [Alias("CN")]  
 [Alias("name","system")]  
 [string[]] $ComputerName )

Test1 "Mars","Saturn" # pass argument by position  
Test1 -ComputerName "Mars","Saturn" # pass argument by name  
Test1 -CN "Mars","Saturn" # pass argument using first alias  
Test1 -name "Mars","Saturn" # pass argument using second alias  
Test1 -sys "Mars","Saturn" # pass argument using third alias

Consider a function call Test2 that has the following param block, and which is called as shown:

param ( [Parameter(Mandatory=$true, ValueFromPipelineByPropertyName=$true)]  
 [Alias('PSPath')]  
 [string] $LiteralPath )  
  
Dir "E:\\*.txt" | Test2 -LiteralPath { $\_ ;"`n`t"; $\_.FullName + ".bak" }  
Dir "E:\\*.txt" | Test2

Cmdlet GetChildItem (alias Dir) adds to the object it returns a new NoteProperty of type string, called PSPath.

### The AllowEmptyCollection attribute

This attribute is used in a script-parameter to allow an empty collection as the argument of a mandatory parameter.

Consider a function call Test that has the following param block, and which is called as shown:

param ( [parameter(Mandatory = $true)]  
 [AllowEmptyCollection()]  
 [string[]] $ComputerName )

Test "Red","Green" # $computerName has Length 2  
Test "Red" # $computerName has Length 1  
Test -comp @() # $computerName has Length 0

### The AllowEmptyString attribute

This attribute is used in a script-parameter to allow an empty string as the argument of a mandatory parameter.

Consider a function call Test that has the following param block, and which is called as shown:

param ( [parameter(Mandatory = $true)]  
 [AllowEmptyString()]  
 [string] $ComputerName )

Test "Red" # $computerName is "Red"  
Test "" # empty string is permitted  
Test -comp "" # empty string is permitted

### The AllowNull attribute

This attribute is used in a script-parameter to allow $null as the argument of a mandatory parameter for which no implicit conversion is available.

Consider a function call Test that has the following param block, and which is called as shown:

param ( [parameter(Mandatory = $true)]  
 [AllowNull()]  
 [int[]] $Values )

Test 10,20,30 # $values has Length 3, values 10, 20, 30  
Test 10,$null,30 # $values has Length 3, values 10, 0, 30  
Test -val $null # $values has value $null

Note that the second case above does not need this attribute; there is already an implicit conversion from $null to int.

### The CmdletBinding attribute

This attribute is used in the attribute-list of param-block of a function to indicate that function acts similar to a cmdlet. Specifically, it allows functions to access a number of methods and properties through the $PsCmdlet variable by using begin, process, and end named blocks (§8.10.6).

When this attribute is present, positional arguments that have no matching positional parameters cause parameter binding to fail and $args is not defined. (Without this attribute $args would take on any unmatched positional argument values.)

The following arguments are used to define the characteristics of the parameter:

| **Parameter Name** | **Purpose** |
| --- | --- |
| SupportsShouldProcess (named) | Type: bool; Default value: $false  Specifies whether the function supports calls to the ShouldProcess method, which is used to prompt the user for feedback before the function makes a change to the system. A value of $true indicates that it does. A value of $false indicates that it doesn't. |
| ConfirmImpact (named) | Type: string; Default value: "Medium"  Specifies the impact level of the action performed. The call to the ShouldProcess method displays a confirmation prompt only when the ConfirmImpact argument is greater than or equal to the value of the $ConfirmPreference preference variable.  The possible values of this argument are:  None: Suppress all requests for confirmation.  Low: The action performed has a low risk of losing data.  Medium: The action performed has a medium risk of losing data.  High: The action performed has a high risk of losing data.  The value of $ConfirmPreference can be set so that only cmdlets with an equal or higher impact level can request confirmation before they perform their operation. For example, if $ConfirmPreference is set to Medium, cmdlets with a Medium or High impact level can request confirmation. Requests from cmdlets with a low impact level are suppressed. |
| DefaultParameterSetName (named) | Type: string; Default value: "\_\_AllParameterSets"  Specifies the parameter set to use if that cannot be determined from the arguments. See the named argument ParameterSetName in the attribute Parameter (§12.3.7). |

Here’s is an example of the framework for using this attribute:

[CmdletBinding(SupportsShouldProcess = $true, ConfirmImpact = "Low")]  
param ( … )  
  
begin { … }  
process { … }  
end { … }

### The OutputType attribute

This attribute is used in the attribute-list of param-block to specify the types returned. The following arguments are used to define the characteristics of the parameter:

| **Parameter Name** | **Purpose** |
| --- | --- |
| Type (position 0) | Type: string[] or array of type literals  A list of the types of the values that are returned. |
| ParameterSetName (named) | Type: string[]  Specifies the parameter sets that return the types indicated by the corresponding elements of the Type parameter. |

Here are several examples of this attribute's use:

[OutputType([int])] param ( … )  
[OutputType("double")] param ( … )  
[OutputType("string","string")] param ( … )

### The Parameter attribute

This attribute is used in a script-parameter. The following named arguments are used to define the characteristics of the parameter:

| **Parameter** | **Purpose** |
| --- | --- |
| HelpMessage (named) | Type: string  This argument specifies a message that is intended to contain a short description of the parameter. This message is used in an implementation-defined manner when the function or cmdlet is run yet a mandatory parameter having a HelpMessage does not have a corresponding argument.  The following example shows a parameter declaration that provides a description of the parameter.  param ( [Parameter(Mandatory = $true,  HelpMessage = "An array of computer names.")]  [string[]] $ComputerName )  Windows PowerShell: If a required parameter is not provided the runtime prompts the user for a parameter value. The prompt dialog box includes the HelpMessage text. |
| Mandatory (named) | Type: bool; Default value: $false  This argument specifies whether the parameter is required within the given parameter set (see ParameterSetName argument below). A value of $true indicates that it is. A value of $false indicates that it isn't.  param ( [Parameter(Mandatory = $true)]  [string[]] $ComputerName )  Windows PowerShell: If a required parameter is not provided the runtime prompts the user for a parameter value. The prompt dialog box includes the HelpMessage text, if any. |
| ParameterSetName (named) | Type: string; Default value: "\_\_AllParameterSets"  It is possible to write a single function or cmdlet that can perform different actions for different scenarios. It does this by exposing different groups of parameters depending on the action it wants to take. Such parameter groupings are called parameter sets.  The argument ParameterSetName specifies the parameter set to which a parameter belongs. This behavior means that each parameter set must have one unique parameter that is not a member of any other parameter set.  For parameters that belong to multiple parameter sets, add a Parameter attribute for each parameter set. This allows the parameter to be defined differently for each parameter set.  A parameter set that contains multiple positional parameters must define unique positions for each parameter. No two positional parameters can specify the same position.  If no parameter set is specified for a parameter, the parameter belongs to all parameter sets.  When multiple parameter sets are defined, the named argument DefaultParameterSetName of the attribute CmdletBinding (§12.3.5) is used to specify the default parameter set. The runtime uses the default parameter set if it cannot determine the parameter set to use based on the information provided by the command, or raises an exception if no default parameter set has been specified.  The following example shows a function Test with a parameter declaration of two parameters that belong to two different parameter sets, and a third parameter that belongs to both sets:  param ( [Parameter(Mandatory = $true,  ParameterSetName = "Computer")]  [string[]] $ComputerName,  [Parameter(Mandatory = $true,  ParameterSetName = "User")]  [string[]] $UserName,  [Parameter(Mandatory = $true,  ParameterSetName = "Computer")]  [Parameter(ParameterSetName = "User")]  [int] $SharedParam = 5 )  if ($PsCmdlet.ParameterSetName -eq "Computer")  {  # handle "Computer" parameter set  }  elseif ($PsCmdlet.ParameterSetName -eq "User")  {  # handle "User" parameter set  }  … }  Test -ComputerName "Mars","Venus" -SharedParam 10 Test -UserName "Mary","Jack" Test -UserName "Mary","Jack" -SharedParam 20 |
| Position (named) | Type: int  This argument specifies the position of the parameter in the argument list. If this argument is not specified, the parameter name or its alias must be specified explicitly when the parameter is set. If none of the parameters of a function has positions, positions are assigned to each parameter based on the order in which they are received.  The following example shows the declaration of a parameter whose value must be specified as the first argument when the function is called.  param ( [Parameter(Position = 0)]  [string[]] $ComputerName ) |
| ValueFromPipeline (named) | Type: bool; Default value: $false  This argument specifies whether the parameter accepts input from a pipeline object. A value of $true indicates that it does. A value of $false indicates that it does not.  Specify $true if the function or cmdlet accesses the complete object, not just a property of the object.  Only one parameter in a parameter set can declare ValueFromPipeline as $true.  The following example shows the parameter declaration of a mandatory parameter, $ComputerName, that accepts the input object that is passed to the function from the pipeline.  param ( [Parameter(Mandatory = $true,  ValueFromPipeline=$true)]  [string[]] $ComputerName )  For an example of using this parameter in conjunction with the Alias attribute see §12.3.1. |
| ValueFromPipelineByPropertyName (named) | Type: bool; Default value: $false  This argument specifies whether the parameter takes its value from a property of a pipeline object that has either the same name or the same alias as this parameter. A value of $true indicates that it does. A value of $false indicates that it does not.  Specify $true if the following conditions are true: the parameter accesses a property of the piped object, and the property has the same name as the parameter, or the property has the same alias as the parameter.  A parameter having ValueFromPipelineByPropertyName set to $true need not have a parameter in the same set with ValueFromPipeline set to $true.  If a function has a parameter $ComputerName, and the piped object has a ComputerName property, the value of the ComputerName property is assigned to the $ComputerName parameter of the function:  param ( [parameter(Mandatory = $true,  ValueFromPipelineByPropertyName = $true)]  [string[]] $ComputerName )  Multiple parameters in a parameter set can define the ValueFromPipelineByPropertyName as $true. Although, a single input object cannot be bound to multiple parameters, different properties in that input object may be bound to different parameters.  When binding a parameter with a property of an input object, the runtime environment first looks for a property with the same name as the parameter.  If such a property does not exist, the runtime environment looks for aliases to that parameter, in their declaration order, picking the first such alias for which a property exists.  function Process-Date {  param(  [Parameter(ValueFromPipelineByPropertyName=$true)]  [int]$Year,  [Parameter(ValueFromPipelineByPropertyName=$true)]  [int]$Month,  [Parameter(ValueFromPipelineByPropertyName=$true)]  [int]$Day  )  process { … } }  Get-Date | Process-Date |
| ValueFromRemainingArguments (named) | Type: bool; Default value: $false  This argument specifies whether the parameter accepts all of the remaining arguments that are not bound to the parameters of the function. A value of $true indicates that it does. A value of $false indicates that it does not.  The following example shows a parameter $others that accepts all the remaining arguments of the input object that is passed to the function Test:  param ( [parameter(Mandatory = $true)][int] $p1,  [parameter(Mandatory = $true)][int] $p2,  [parameter(ValueFromRemainingArguments = $true)]  [string[]] $others )  Test 10 20 # $others has Length 0 Test 10 20 30 40 # $others has Length 2, value 30,40 |

An implementation may define other attributes as well.

Windows PowerShell: The following attributes are provided as well:

HelpMessageBaseName: Specifies the location where resource identifiers reside. For example, this parameter could specify a resource assembly that contains Help messages that are to be localized.

HelpMessageResourceId: Specifies the resource identifier for a Help message.

### The ValidateCount attribute

This attribute is used in a script-parameter to specify the minimum and maximum number of argument values that the parameter can accept. The following arguments are used to define the characteristics of the parameter:

| **Parameter Name** | **Purpose** |
| --- | --- |
| MinLength (position 0) | Type: int  This argument specifies the minimum number of argument values allowed. |
| MaxLength (position 1) | Type: int  This argument specifies the maximum number of argument values allowed. |

In the absence of this attribute, the parameter's corresponding argument value list can be of any length.

Consider a function call Test that has the following param block, and which is called as shown:

param ( [ValidateCount(2,5)]  
 [int[]] $Values )

Temp 10,20,30  
Temp 10 # too few argument values  
Temp 10,20,30,40,50,60 # too many argument values

### The ValidateLength attribute

This attribute is used in a script-parameter to specify the minimum and maximum length of the parameter's argument, which must have type string. The following arguments are used to define the characteristics of the parameter:

| **Parameter Name** | **Purpose** |
| --- | --- |
| MinLength (position 0) | Type: int  This argument specifies the minimum number of characters allowed. |
| MaxLength (position 1) | Type: int  This argument specifies the maximum number of characters allowed. |

In the absence of this attribute, the parameter's corresponding argument can be of any length.

Consider a function call Test that has the following param block, and which is called as shown:

param ( [parameter(Mandatory = $true)]  
 [ValidateLength(3,6)]  
 [string[]] $ComputerName )

Test "Thor","Mars" # lengths ok  
Test "Io","Mars" # "Io" is too short  
Test "Thor","Jupiter" # "Jupiter" is too long

### The ValidateNotNull attribute

This attribute is used in a script-parameter to specify that the argument of the parameter cannot be $null or be a collection containing a $null-valued element.

Consider a function call Test that has the following param block, and which is called as shown:

param ( [ValidateNotNull()]  
 [string[]] $Names )

Test "Jack","Jill" # ok  
Test "Jane",$null # $null array element value not allowed   
Test $null # null array not allowed

### The ValidateNotNullOrEmpty attribute

This attribute is used in a script-parameter to specify that the argument if the parameter cannot be $null, an empty string, or an empty array, or be a collection containing a $null-valued or empty string element.

Consider a function call Test that has the following param block, and which is called as shown:

param ( [ValidateNotNullOrEmpty()]  
 [string[]] $Names )

Test "Jack","Jill" # ok  
Test "Mary","" # empty string not allowed   
Test "Jane",$null # $null array element value not allowed   
Test $null # null array not allowed  
Test @() # empty array not allowed

### The ValidatePattern attribute

This attribute is used in a script-parameter to specify a regular expression for matching the pattern of the parameter's argument. The following arguments are used to define the characteristics of the parameter:

| **Parameter Name** | **Purpose** |
| --- | --- |
| RegexString (position 0) | Type: String  A regular expression that is used to validate the parameter's argument |
| Options (named) | Type: Regular-Expression-Option  See §4.2.6.4 for the allowed values. |

If the argument is a collection, each element in the collection must match the pattern.

Consider a function call Test that has the following param block, and which is called as shown:

param ( [ValidatePattern('^[A-Z][1-5][0-9]$')]  
 [string] $Code,

[ValidatePattern('^(0x|0X)([A-F]|[a-f]|[0-9])([A-F]|[a-f]|[0-9])$')]  
 [string] $HexNum,

[ValidatePattern('^[+|-]?[1-9]$')]  
 [int] $Minimum )

Test -c A12 # matches pattern  
Test -c A63 # does not match pattern

Test -h 0x4f # matches pattern  
Test -h "0XB2" # matches pattern  
Test -h 0xK3 # does not match pattern

Test -m -4 # matches pattern  
Test -m "+7" # matches pattern  
Test -m -12 # matches pattern, but is too long

### The ValidateRange attribute

This attribute is used in a script-parameter to specify the minimum and maximum values of the parameter's argument. The following arguments are used to define the characteristics of the parameter:

| **Parameter Name** | **Purpose** |
| --- | --- |
| MinRange (position 0) | Type: object  This argument specifies the minimum value allowed. |
| MaxRange (position 1) | Type: object  This argument specifies the maximum value allowed. |

In the absence of this attribute, there is no range restriction.

Consider a function call Test1 that has the following param block, and which is called as shown:

param ( [parameter(Mandatory = $true)]  
 [ValidateRange(1,10)]  
 [int] $StartValue )

Test1 2  
Test1 -st 7  
Test1 -3 # value is too small  
Test1 12 # value is too large

Consider a function call Test2 that has the following param block and calls:

param ( [parameter(Mandatory = $true)]  
 [ValidateRange("b","f")]  
 [string] $Name )

Test2 "Bravo" # ok  
Test2 "Alpha" # value compares less than the minimum  
Test2 "Hotel" # value compares greater than the maximum

Consider a function call Test3 that has the following param block, and which is called as shown:

param ( [parameter(Mandatory = $true)]  
 [ValidateRange(0.002,0.003)]  
 [double] $Distance )

Test3 0.002  
Test3 0.0019 # value is too small  
Test3 "0.005" # value is too large

### The ValidateScript attribute

This attribute is used in a script-parameter to specify a script that is to be used to validate the parameter's argument.

The argument in position 1 is a script-block-expression.

Consider a function call Test that has the following param block, and which is called as shown:

param ( [Parameter(Mandatory = $true)]  
 [ValidateScript({($\_ -ge 1 -and $\_ -le 3) -or ($\_ -ge 20)})]  
 [int] $Count )

Test 2 # ok, valid value  
Test 25 # ok, valid value  
Test 5 # invalid value  
Test 0 # invalid value

### The ValidateSet attribute

This attribute is used in a script-parameter to specify a set of valid values for the argument of the parameter. The following arguments are used to define the characteristics of the parameter:

| **Parameter Name** | **Purpose** |
| --- | --- |
| ValidValues (position 0) | Type: string[]  The set of valid values. |
| IgnoreCase (named) | Type: bool; Default value: $true  Specifies whether case should be ignored for parameters of type string. |

If the parameter has an array type, every element of the corresponding argument array must match an element of the value set.

Consider a function call Test that has the following param block, and which is called as shown:

param ( [ValidateSet("Red","Green","Blue")]  
 [string] $Color,

[ValidateSet("up","down","left","right", IgnoreCase = $false)]  
 [string] $Direction

)

Test -col "RED" # case is ignored, is a member of the set   
Test -col "white" # case is ignored, is not a member of the set

Test -dir "up" # case is not ignored, is a member of the set  
Test -dir "Up" # case is not ignored, is not a member of the set

# Cmdlets

A cmdlet is a single-feature command that manipulates objects in PowerShell. Cmdlets can be recognized by their name format, a verb and noun separated by a dash (-), such as Get-Help, Get-Process, and Start-Service. A verb pattern is a verb expressed using wildcards, as in W\*. A noun pattern is a noun expressed using wildcards, as in \*event\*.

Cmdlets should be simple and be designed to be used in combination with other cmdlets. For example, "get" cmdlets should only retrieve data, "set" cmdlets should only establish or change data, "format" cmdlets should only format data, and "out" cmdlets should only direct the output to a specified destination.

[Implementer's Note: For each cmdlet provide a help file that can be accessed by typing:

get-help cmdlet-name -detailed

The detailed view of the cmdlet help file should include a description of the cmdlet, the command syntax, descriptions of the parameters, and an example that demonstrate the use of that cmdlet. end note]

Cmdlets are used similarly to operating system commands and utilities. PowerShell commands are not case-sensitive.

## Add-Content (alias ac)

Synopsis:

Adds content to the specified items (§3.3).

Syntax:

Add-Content [ -LiteralPath ] <string[]> [ -Value <object[]> ]  
[ -Credential <Credential> ] [ -Exclude <string[]> ] [ -Filter <string> ]  
[ -Force ] [ -Include <string[]> ] [ -PassThru ] [ -Confirm ] [ -WhatIf ]  
[ <CommonParameters> ]

Add-Content [ -Path ] <string[]> [ -Value <object[]> ] [ -Credential <Credential> ]  
[ -Exclude <string[]> ] [ -Filter <string> ] [ -Force ] [ -Include <string[]> ]  
[ -PassThru ] [ -Confirm ] [ -WhatIf ] [ <CommonParameters> ]

Description:

This cmdlet appends content to a specified item.

Parameters:

-Confirm [ <SwitchParameter> ] — (alias cf) Prompts for confirmation before executing the cmdlet. If confirmation is denied, the cmdlet terminates with the new variable being created or changed.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Credential <Credential> — Specifies a user account that has permission to perform this action. The default is the current user. See §4.5.23.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-Exclude <string[]> — Specifies the path(s) to be excluded from the operation.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Filter <string> — Specifies a file filter in the provider's format or language. This parameter qualifies the Path parameter. The syntax of the filter, including the use of wildcards, depends on the provider.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Force [ <SwitchParameter> ] — This switch parameter allows the cmdlet to write over an existing item. However, this parameter cannot be used to override security restrictions.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Include <string[]> — Specifies the path(s) to be included in the operation.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-LiteralPath <string[]> — (alias PSPath) Specifies the path(s) of the item. The strings are used exactly as they are written. Characters that look like wildcards are not interpreted as such.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-PassThru [ <SwitchParameter> ] — Returns an object representing the added content. By default, this cmdlet does not generate any output.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Path <string[]> — Specifies the path(s) of the item.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Value <object[]> — Specifies the values to be added.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByValue or ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-WhatIf [ <SwitchParameter> ] — (alias wi) Describes what would happen if the cmdlet were executed without actually executing it.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

For the Value parameter, one or more objects of any kind can be written to the pipeline. However, the object is converted to a string before it is added to the item.

Outputs:

None unless the PassThru switch parameter is used.

Examples:

Add-Content "J:\Test\File2.txt" -Value "`nLine 5"

## Add-Member

Synopsis:

Adds a user-defined custom member to an instance of a PowerShell object.

Syntax:

Add-Member [ -MemberType ] { AliasProperty | NoteProperty | ScriptProperty  
| PropertySet | ScriptMethod | MemberSet } [ -Name ] <string>  
-InputObject <object> [ [ -Value ] <object> ] [ [ -SecondValue ] <object>]  
[ -Force ] [ -PassThru ] [ <CommonParameters>]

Description:

This cmdlet adds a user-defined custom member to an instance of a PowerShell object. The following types of members can be added: AliasProperty, NoteProperty, ScriptProperty, PropertySet, ScriptMethod, and MemberSet. The initial value of the member can be set via the Value parameter. For AliasProperty and ScriptProperty, additional information can be supplied via the SecondValue parameter.

The new member is added to the object piped to this cmdlet or the one specified via the InputObject parameter. The additional member is available only while that instance exists.

Windows PowerShell: The following extra member types are permitted: CodeProperty and CodeMethod.

The object (including the addition member) can be saved to disk using the cmdlet Export-Clixml.

Parameters:

-Force [ <SwitchParameter> ] — This switch parameter causes a new member to be added even if one with the same name (that was previously added by Add-Member) already exists.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-InputObject <object> — The object to which the new member is added.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByValue (§12.3.7) | | Accept wildcard characters: No |

-MemberType <PsMemberTypes> —Specifies the type of the member to add.

* AliasProperty: A new name for an existing property.
* MemberSet: A predefined collection of properties and methods, such as PsBase and PsTypeNames (§4.6).
* NoteProperty: A property with a static value.
* PropertySet: A predefined collection of object properties.
* ScriptMethod: A method whose value is the output of a script.
* ScriptProperty: A property whose value is the output of a script.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: none |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Name <string> — Specifies the name of the new member.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 2 | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-PassThru [ <SwitchParameter> ] — Passes the newly extended object to the pipeline.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-SecondValue <object> — Specifies optional additional information. If used when adding an AliasProperty, this parameter must be a data type. A conversion to the specified data type is added to the value of the AliasProperty. (For example, if an AliasProperty is added that provides an alternate name for a string property, a SecondValue parameter of int might be specified to indicate that the value of that string property should be converted to an integer.) If used when adding a ScriptProperty, this parameter specifies an additional ScriptBlock. In that case, the ScriptBlock specified in the Value parameter is used to get the value of a variable. The ScriptBlock specifies in the SecondValue parameter is used to set the value of a variable.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Position 4 | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Value <object> — Specifies the value of the new member, or for MemberSet or PropertySet, specifies a collection of members.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Position 3 | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

Any object can be piped to this cmdlet.

Outputs:

None unless the PassThru switch parameter is used, in which case, the newly extended object is returned.

Examples:

(get-childitem)[0] | Add-Member -MemberType NoteProperty -Name Status  
 -Value done  
(get-childitem)[0] | Add-Member -MemberType AliasProperty -Name FileLength" -Value Length  
$a = "a string"  
$a = $a | Add-Member -MemberType NoteProperty -Name StringUse -Value Display  
 -Passthru

## Clear-Content (alias clc)

Synopsis:

Deletes the contents of one or more items (§3.3), but does not delete them.

Syntax:

Clear-Content [ -LiteralPath ] <string[]> [ -Credential <Credential> ]  
[ -Exclude <string[]> ] [ -Filter <string> ] [ -Force ] [ -Include <string[]> ]  
[ -Confirm ] [ -WhatIf ] [ <CommonParameters> ]

Clear-Content [ -Path ] <string[]> [ -Credential <Credential> ]  
[ -Exclude <string[]> ] [ -Filter <string> ] [ -Force ] [ -Include <string[]> ]  
[ -Confirm ] [ -WhatIf ] [ <CommonParameters> ]

Description:

This cmdlet deletes the contents of one or more items, but it does not delete them. Clear-Content is similar to Clear-Item, but Clear-Content works on files instead of on aliases and variables.

Parameters:

-Confirm [ <SwitchParameter> ] — (alias cf) Prompts for confirmation before executing the cmdlet. If confirmation is denied, the cmdlet terminates with the new variable being created or changed.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Credential <Credential> — Specifies a user account that has permission to perform this action. The default is the current user. See §4.5.23

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-Exclude <string[]> — Specifies the path(s) to be excluded from the operation.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: Yes |

-Filter <string> — Specifies a file filter in the provider's format or language. This parameter qualifies the Path parameter. The syntax of the filter, including the use of wildcards, depends on the provider.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Force [ <SwitchParameter> ] — Allows the file contents to be cleared even if the file is read-only.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Include <string[]> — Specifies the path(s) to be included in the operation.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: Yes |

-LiteralPath <string[]> — (alias PSPath) Specifies the path(s) of the item(s). The strings are used exactly as they are written. Characters that look like wildcards are not interpreted as such.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Path <string[]> — Specifies the path(s) of the item(s).

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-WhatIf [ <SwitchParameter> ] — (alias wi) Describes what would happen if the cmdlet were executed without actually executing it.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

A path (but not literal path) can be written to the pipeline to Clear-Content.

Outputs:

None

Examples:

Clear-Content J:\Files\Test\*.\*

## Clear-Item (alias cli)

Synopsis:

Deletes the contents of one or more items (§3.3), but does not delete them.

Syntax:

Clear-Item [ -LiteralPath ] <string[]> [ -Credential <Credential> ]  
[ -Exclude <string[]> ] [ -Filter <string> ] [ -Force ] [ -Include <string[]> ]  
[ -Confirm ] [ -WhatIf ] [ <CommonParameters> ]

Clear-Item [ -Path ] <string[]> [ -Credential <Credential> ]  
[ -Exclude <string[]> ] [ -Filter <string> ] [ -Force ] [ -Include <string[]> ]  
[ -Confirm ] [ -WhatIf ] [ <CommonParameters> ]

Description:

This cmdlet deletes the value of one or more items, but it does not delete them.

Parameters:

-Confirm [ <SwitchParameter> ] — (alias cf) Prompts for confirmation before executing the cmdlet. If confirmation is denied, the cmdlet terminates with the new variable being created or changed.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Credential <Credential> — Specifies a user account that has permission to perform this action. The default is the current user. See §4.5.23

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-ExcludePath <string[]> — Specifies the path(s) to exclude from the operation.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Filter <string> — Specifies a file filter in the provider's format or language. This parameter qualifies the Path parameter. The syntax of the filter, including the use of wildcards, depends on the provider.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Force [ <SwitchParameter> ] — Allows the cmdlet to clear items that cannot otherwise be changed, such as read- only aliases. The cmdlet cannot clear constants. Implementation varies from provider to provider.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-IncludePath <string[]> — Specifies the path(s) to include in the operation.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-LiteralPath <string[]> — (alias PSPath) Specifies the path(s) to the item(s) to be cleared. The strings are used exactly as they are written. Characters that look like wildcards are not interpreted as such.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Path <string[]> — Specifies the path(s) to the item(s) to be cleared.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByValue or ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-WhatIf [ <SwitchParameter> ] — (alias wi) Describes what would happen if the cmdlet were executed without actually executing it.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

A path (but not literal path) can be written to the pipeline to Clear-Item.

Outputs:

None

Examples:

Clear-Item Variable:Count  
Clear-Item Alias:log\* -Include \*1\* -Exclude \*3\*

## Clear-Variable (alias clv)

Synopsis:

Deletes the value of one or more existing variables.

Syntax:

Clear-Variable [ -Name ] <string[]> [ -Exclude *<string[]>* ] [ -Force ]  
[ -Include *<string[]>* ] [ -PassThru ] [ -Scope <string> ]   
[ -Confirm ] [ -WhatIf ] [ <CommonParameters> ]

Description:

This cmdlet deletes the value of one or more existing variables, but the variable itself is not deleted.

Parameters:

-Confirm [ <SwitchParameter> ] — (alias cf) Prompts for confirmation before executing the cmdlet. If confirmation is denied, the cmdlet terminates with the new variable being created or changed.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Exclude <string[]> — Specifies the name(s) to be excluded from the operation.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: Yes |

-Force [ <SwitchParameter> ] — This switch parameter allows the value of an existing read-only variable to be deleted.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Include <string[]> — Specifies the name(s) to be included in the operation.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: Yes |

-Name <string[]> — Specifies the name(s) of the variables whose values are to be deleted.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-PassThru [ <SwitchParameter> ] — Causes the cmdlet to write to the pipeline an object that represents the cleared variable (§4.5.3). (Ordinarily, the cmdlet does not write any output to the pipeline.)

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Scope <string> — Specifies the scope (§3.5) of the variable whose value is to be deleted. The valid values are "Global", "Script", and "Local", or scope number (§3.5.2).

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: "Local" |
| Accept pipeline input: No | | Accept wildcard characters: No |

-WhatIf [ <SwitchParameter> ] — (alias wi) Describes what would happen if the cmdlet were executed without actually executing it.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

None.

Outputs:

None unless the PassThru switch parameter is used.

Examples:

Clear-Variable 'Count10?' -Exclude 'Count101','Count102'

## Compare-Object (alias compare)

Synopsis:

Compares two sets of objects.

Syntax:

Conpare-Object [ -ReferenceObject ] <object[]> ]  
[ -DifferenceObject ] <object[]> ] [ -CaseSensitive ]  
[ -Culture <string> ] [ -ExcludeDifferent ] [ -IncludeEqual ] [ -PassThru ]  
[ -Property <object[]> ] [ -SyncWindow <int> ] [ CommonParameters> ]

Description:

This cmdlet compares two sets of objects, the Reference set, and the Difference set. The result of the comparison indicates whether a property value appeared only in the object from the Reference set, only in the object from the Difference set or, if the IncludeEqual parameter is specified, in both objects.

Parameters:

-CaseSensitive [ <SwitchParameter> ] — Indicates that the comparisons should be case sensitive. By default, the comparisons are not case sensitive.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Culture <string> — Specifies the cultural configuration to use for comparisons (as in "en-US" for US English using language codes from ISO 639-1 and country codes from ISO 3166-1).

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-DifferenceObject <object[]> — Specifies the objects that are compared to the reference objects.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 2 | Default value: None |
| Accept pipeline input: Yes, ByPropertyValue (§12.3.7) | | Accept wildcard characters: No |

-ExcludeDifferent [ <SwitchParameter> ] — Outputs only the characteristics of compared objects that are equal.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-IncludeEqual [ <SwitchParameter> ] — Outputs characteristics of compared objects that are equal. By default, only characteristics that differ between the reference and difference objects are output.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-PassThru [ <SwitchParameter> ] — Passes to the pipeline the objects that differed.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-ReferenceObject <object[]> — Objects used as a reference for comparison.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Property <object[]> — Specifies the properties of the reference and difference objects to compare.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-SyncWindow <int> — Defines a search region in which an attempt is made to re-synchronize the order if there is no match.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: [int]::MaxValue |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

The DifferenceObject object set can be piped to this cmdlet.

Outputs:

Depends on the use of the PassThru, ExcludeDifferent, and IncludeEqual parameters.

Examples:

File1.txt contains the records red, green, yellow, blue, and black. File1.txt contains the records red, Green, blue, Black, and white.

Example 1:

Compare-Object -ReferenceObject $(Get-Content File1.txt) -DifferenceObject  
 $(Get-Content File2.txt) -PassThru

The output produced is:

White  
yellow

Example 2:

Compare-Object -ReferenceObject $(Get-Content File1.txt) -DifferenceObject  
 $(Get-Content File2.txt) -PassThru -CaseSensitive

The output produced is:

Green  
Black  
white  
green  
yellow  
black

Example 3:

Compare-Object -ReferenceObject $(Get-Content File1.txt) -DifferenceObject   
 $(Get-Content File2.txt) -PassThru -ExcludeDifferent -IncludeEqual

The output produced is:

red  
green  
blue  
black

## ConvertFrom-StringData

Synopsis:

Converts a string containing one or more key/value pairs to a hash table.

Syntax:

ConvertFrom-StringData [ -StringData ] <string> [ <CommonParameters> ]

Description:

This cmdlet converts a string that contains one or more key/value pairs into a hash table.

This cmdlet can be called from within the statement-block of a data section (§8.9).

Parameters:

-StringData <string> — Specifies the string to be converted. Each key/value pair must be on a separate line, or each pair must be separated by a newline character (`n).

If the string contains multiple lines, any line that does not define a key/value pair may contain text that looks like a single-line-comment. Such comment-like lines are ignored.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyValue (§12.3.7) | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

A string containing a key/value pair to can be written to the pipeline to ConvertFrom-StringData.

Outputs:

The hash table that is created from the key/value pairs.

Examples:

$str = @'  
Msg1 = The string parameter is required.  
Msg2 = Credentials are required for this command.  
Msg3 = The specified variable does not exist.  
'@  
$v = ConvertFrom-StringData -StringData $str

$v = $str | ConvertFrom-StringData

$v = ConvertFrom-StringData -StringData @'  
 Name = Disks.ps1  
 # Category is optional.  
 Category = Storage  
 Cost = Free  
'@  
$v = ConvertFrom-StringData -StringData "Top = Red `n Bottom = Blue"

## Convert-Path (alias cvpa)

Synopsis:

Converts a path (§3.4) from a PowerShell path to a PowerShell provider path.

Syntax:

Convert-Path [ -LiteralPath ] <string[]> [ <CommonParameters> ]

Convert-Path [ -Path ] <string[]> [ <CommonParameters> ]

Description:

This cmdlet converts a path from a PowerShell path to a PowerShell provider path.

Parameters:

-LiteralPath <string[]> — (alias PSPath) Specifies the path(s) to be converted. The strings are used exactly as they are written. Characters that look like wildcards are not interpreted as such.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-Path <string[]> — Specifies the path(s) to be converted.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByName or ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

A string containing a path (but not a child path) can be written to the pipeline to Convert-Path.

Outputs:

If the result is one path, that string is output. Otherwise, an unconstrained 1-dimensional array of string values is output, whose elements correspond to the path results.

Examples:

Convert-Path -Path E:.  
Convert-Path -Path E:.,G:\Temp\..

## Copy-Item (alias copy, cp, cpi)

Synopsis:

Copies one or more items (§3.3) from one location to another.

Syntax:

Copy-Item [ -LiteralPath ] <string[]> [ [ -Destination ] <string> ]  
[ -Container ] [ -Credential <Credential> ] [ -Exclude <string[]> ]  
[ -Filter <string> ] [ -Force ] [ -Include <string[]> ] [ -PassThru ]  
[ -Recurse ] [ -Confirm ] [ -WhatIf ] [ <CommonParameters> ]

Copy-Item [ -Path ] <string[]> [ [ -Destination ] <string> ] [ -Container ]  
[ -Credential <Credential> ] [ -Exclude <string[]> ] [ -Filter <string> ]  
[ -Force ] [ -Include <string[]> ] [ -PassThru ] [ -Recurse ] [ -Confirm ]  
[ -WhatIf ] [ <CommonParameters> ]

Description:

This cmdlet copies one or more items from one location to another. The particular items that the cmdlet can copy depend on the providers available.

Parameters:

-Confirm [ <SwitchParameter> ] — (alias cf) Prompts for confirmation before executing the cmdlet. If confirmation is denied, the cmdlet terminates with the new variable being created or changed.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Container [ <SwitchParameter> ] — Preserves container objects during the copy operation.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Credential <Credential> — Specifies a user account that has permission to perform this action. The default is the current user. See §4.5.23

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-Destination <string> — Specifies the destination path of the copy.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 2 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-ExcludePath <string[]> — Specifies the path(s) to exclude from the operation.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Filter <string> — Specifies a file filter in the provider's format or language. This parameter qualifies the Path parameter. The syntax of the filter, including the use of wildcards, depends on the provider.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Force [ <SwitchParameter> ] — Allows the cmdlet to copy items that cannot otherwise be changed, such as copying over a read-only file or alias.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-IncludePath <string[]> — Specifies the path(s) to include in the operation.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-LiteralPath <string[]> — (alias PSPath) Specifies the path(s) to the item(s) to be copied. The strings are used exactly as they are written. Characters that look like wildcards are not interpreted as such.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-PassThru [ <SwitchParameter> ] — Returns one or more objects representing the copied items. By default, this cmdlet does not generate any output.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Path <string[]> — Specifies the path(s) to the item(s) to be copied.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByValue or ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Recurse [ <SwitchParameter> ] — Specifies a recursive copy.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-WhatIf [ <SwitchParameter> ] — (alias wi) Describes what would happen if the cmdlet were executed without actually executing it.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

A path (but not literal path) can be written to the pipeline to Copy-Item.

Outputs:

If PassThru is present, a single object (§4.5) or an unconstrained 1-dimensional array of objects that describes the item(s) copied.

Examples:

Copy-Item -Path "J:\Test\File2.txt" -Destination "J:\Test\File2-v2.txt"  
Copy-Item -Path "J:\Test\\*","J:\Test3\\*" -Destination "J:\" -PassThru  
Copy-Item -Path "E:\Temp\\*" -Destination "J:\Test" -Container -Recurse

## Export-Alias (alias epal)

Synopsis:

Exports alias information to a file.

Syntax:

Export-Alias [ -Path ] <string> [ [ -Name ] <string> ] [ [ -Append ]  
[ -As { Csv | Script } ] [ -Description <string> ] [ -Force ] [ -NoClobber ]  
[ -PassThru ] [ -Scope <string> ] [ -Confirm ] [ -WhatIf ] [ <CommonParameters> ]

Description:

This cmdlet exports alias information to a file, allowing the scope and file format to be specified.

Parameters:

-Append [ <SwitchParameter> ] — This switch parameter causes the output to be appended if the output file exists; otherwise, it causes the output file to be created.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-As <ExportAliasFormat> — Sets the format of the output file. The valid values are:

* Csv: Use comma-separated value format.
* Script: Write a Set-Alias command for each alias exported, such that the resulting file is suitable for direct use as a script file.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: "Csv" |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Confirm [ <SwitchParameter> ] — (alias cf) Prompts for confirmation before executing the cmdlet. If confirmation is denied, the cmdlet terminates with the new variable being created or changed.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Description <string> — Adds a description to the exported file as a comment at the top of the file, following the header information comments.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Force [ <SwitchParameter> ] — This switch parameter causes the output file to be overridden if it is marked read-only. If both Force and NoClobber are used, NoClobber takes precedence.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Name <string> — Specifies the names of the aliases to be exported.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Position 1 | Default value: Export all aliases |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-NoClobber [ <SwitchParameter> ] — This switch parameter prevents Export-Alias from overwriting any files, even if Force is present. If NoClobber is omitted, Export-Alias overwrites an existing file without warning, unless the read-only attribute is set on the file. NoClobber does not prevent -Append from adding content to an existing file. If both Force and NoClobber are used, NoClobber takes precedence.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: Overwrites read-write files |
| Accept pipeline input: No | | Accept wildcard characters: No |

-PassThru [ <SwitchParameter> ] — Causes the cmdlet to write to the pipeline an object that represents each alias (§4.5.4) written to the file. (Ordinarily, the cmdlet does not write any output to the pipeline.)

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Path <string> — (alias PSPath) Specifies the path of the file to be written to. Although wildcards are permitted, they must resolve to a single file.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Scope <string> — Specifies the scope (§3.5) of the aliases to be exported. The valid values are "Global", "Script", and "Local", or scope number (§3.5.2).

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: "Local" |
| Accept pipeline input: No | | Accept wildcard characters: No |

-WhatIf [ <SwitchParameter> ] — (alias wi) Describes what would happen if the cmdlet were executed without actually executing it.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

None.

Outputs:

None unless the PassThru switch parameter is used.

Examples:

Export-Alias 'E:\Powershell\Scripts\AliasList.txt' -As Csv  
Export-Alias 'E:\Powershell\Scripts\Locals.ps1' -As Script -Scope "Local"

## Export-ModuleMember

Synopsis:

Identifies the module (§3.14) members that are to be exported.

Syntax:

Export-ModuleMember [ [ -Function ] <string[]> ] [ -Alias <string[]> ]  
[ -Cmdlet <string[]> ] [ -Variable <string[]> ] [ <CommonParameters> ]

Description:

This cmdlet specifies the module members that are to be exported from a script module or from a dynamic module. (This cmdlet can be used only in a script module file or a dynamic module.)

If a script module does not include an Export-ModuleMember command, the functions in the script module are exported, but no names are. When a script module includes one or more Export-ModuleMember commands, only the members specified in the Export-ModuleMember commands are exported.

Export-ModuleMember can be used to export members that the script module imports from other modules.

Parameters:

-Alias <string[]> — Specifies the aliases that are to be exported from the module.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Cmdlet <string[]> — Specifies the cmdlets that are to be exported from the module. (Although cmdlets cannot be defined in a script module, they can be imported into one from a binary module, and then re-exported.)

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Function <string[]> — Specifies the functions that are to be exported from the module.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByValue or ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Variable <string[]> — Specifies the variables that are to be exported from the module.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

Function names can be written to the pipeline to Export-ModuleMember.

Outputs:

None

Examples:

Export-ModuleMember -Function CToF,FToC -Alias c2f,f2c  
Export-ModuleMember -Variable boolingTempC,boolingTempF  
Export-ModuleMember -Variable freezingTempC,freezingTempF

## ForEach-Object (alias %, foreach)

Synopsis:

Performs an operation against each of a set of input objects.

Syntax:

ForEach-Object [ -Process ] <scriptblock[]> [ -Begin < scriptblock> ]  
[ -End < scriptblock> ] [ -InputObject <object> ] [ <CommonParameters> ]

Description:

This cmdlet performs one or more operations on each of a set of input objects. These operations are described within one or more script blocks provided as the value of the Process parameter. Within these script blocks, the current input object is represented by the variable $\_.

In addition to using the script block that describes the operations to be carried out on each input object, two additional script blocks can be provided. One, specified as the value of the Begin parameter, runs before the first input object is processed. The other, specified as the value of the End parameter, runs after the last input object is processed.

The results of the evaluation of all the script blocks are passed down the pipeline.

If the Begin and End parameters are omitted, then, if the Process parameter contains only one block, it is interpreted as the process block (§8.10.6). If it contains two blocks, the first is taken as the begin block and the second as the process block. If it contains three blocks, they represent the begin, process, and end blocks, in that order. If it contains more than three blocks, the first and last correspond to the begin and end blocks, respectively, and the rest are collectively taken to be the set of process blocks.

If the Begin and Process parameters are both present, the first block specified by Process is the first process block, not a begin block. If the End and Process parameters are both present, the last block specified by Process is the last process block, not an end block.

Parameters:

-Begin <scriptblock> — Specifies a script block to run after processing all input objects (see §8.10.5).

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-End <scriptblock> — Specifies a script block to run before processing any input objects (see §8.10.5).

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-InputObject <object> — The object on which the Process parameter script block operates.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByValue (§12.3.7) | | Accept wildcard characters: No |

-Process <scriptblock[]> — Specifies one or more script blocks to be applied to each incoming object.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

Any object can be written to the pipeline in place of an InputObject parameter value.

Outputs:

The cumulative output produced by the Begin, Process, and End script blocks, in that order.

Examples:

5,20,-3 | ForEach-Object -Process {$\_ \* 2}  
5,20,-3 | % -Begin { "Setup" } -Process {$\_ \* $\_} -End { "Cleanup" }

## Get-Alias (alias gal)

Synopsis:

Gets alias information.

Syntax:

Get-Alias [ -Name ] <string[]> [ -Exclude <string[]> ] [ -Scope <string> ]  
[ <CommonParameters> ]

Get-Alias [ -Definition <string[]> ] [ -Exclude <string[]> ] [ -Scope <string> ]  
[ <CommonParameters> ]

Description:

Using the first form above, this cmdlet gets information about the specified aliases. Using the second form above, this cmdlet gets information about the aliases for one or more command or command elements.

Parameters:

-Definition <string[]> — Specifies the name(s) of one or more command or command elements whose alias information is to be retrieved.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: Yes |

-Exclude <string[]> — Specifies the name(s) to be excluded from the operation.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: Yes |

-Name <string[]> — Specifies the aliases whose information is to be retrieved.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByValue or ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Scope <string> — Specifies the scope (§3.5) of the new alias. The valid values are "Global", "Script", and "Local", or scope number (§3.5.2).

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: "Local" |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

One or more alias names can be given to this cmdlet.

Outputs:

An object that represents each alias retrieved (§4.5.4).

Examples:

Get-Alias -Name "Fun\*"  
Get-Alias -Definition F1,F2

## Get-ChildItem (alias dir, gci, ls)

Synopsis:

Gets the items (§3.3) and child items at one or more specified locations.

Syntax:

Get-ChildItem [ [ -Path ] <string[]> ] [ [ -Filter ] <string> ] [ -Exclude <string[]> ]  
[ -Force ] [ -Include <string[]> ] [ -Name ] [ -Recurse ] [ <CommonParameters> ]

Get-ChildItem [ [ -LiteralPath ] <string[]> ] [ [ -Filter ] <string> ]  
[ -Exclude <string[]> ] [ -Force ] [ -Include <string[]> ] [ -Name ] [ -Recurse ]  
[ <CommonParameters> ]

Description:

This cmdlet gets the items in one or more specified locations. If the item is a container, the cmdlet gets the items inside the container, known as child items. The Recurse parameter provides access to the items in all child containers.

-Exclude <string[]> — Specifies the path(s) to exclude from the operation.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: Yes |

-Filter <string> — Specifies a file filter in the provider's format or language. This parameter qualifies the Path parameter. The syntax of the filter, including the use of wildcards, depends on the provider.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 2 | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: Yes |

-Force [ <SwitchParameter> ] — Allows the cmdlet to get items that cannot otherwise be accessed by the user, such as hidden or system files. Implementation varies from provider to provider.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Include <string[]> — Specifies the path(s) to include in the operation.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: Yes |

-LiteralPath <string[]> — (alias PSPath) Specifies the path(s) to the item(s). The strings are used exactly as they are written. Characters that look like wildcards are not interpreted as such.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Name [ <SwitchParameter> ] — Retrieves the names of the items, not the items themselves.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Path <string[]> — Specifies the path(s) to the item(s).

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByValue or ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Recurse [ <SwitchParameter> ] — Gets the items in the specified locations and in all child items of the locations.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

A path (but not a literal path) can be written to the pipeline to Get-ChildItem.

Outputs:

When Name is specified, a single string or an unconstrained 1-dimensional array of string containing the name(s) of the item(s) found.

When Name is not specified, a single object (§4.5) or an unconstrained 1-dimensional array of objects that describes the item(s) found.

The object returned contains a new NoteProperty of type string, called PSPath.

Examples:

Get-ChildItem "J:\Test\File2.txt"  
Get-ChildItem "J:\Test\File2.txt" -Name  
Get-ChildItem "J:\F\*.\*" -Recurse

## Get-Command (alias gcm)

Synopsis:

This cmdlet gets information about cmdlets and other elements of commands.

Syntax:

Get-Command [ [ -Name ] <string[]> ] [ [ -ArgumentList ] <object[]> ]  
[ -CommandType { Alias | Function | Filter | Cmdlet | ExternalScript  
| Application | Script | All } ] [ -Module <string[]> ] [ -Syntax ]  
[ -TotalCount <int> ] [ <CommonParameters> ]

Get-Command [ [ -ArgumentList ] <object[]> ] [ -Module <string[]> ]  
[ -Noun <string[]> ] [ -Syntax ] [ -TotalCount <int> ] [ -Verb <string[]> ]  
[ <CommonParameters> ]

Description:

This cmdlet gets information about cmdlets and other elements of commands in the current session, such as aliases, functions, filters, scripts, and applications.

Parameters:

-ArgumentList <object[]> — (alias Args) Gets information about a cmdlet or function when it is used with the specified parameters.

To detect dynamic parameters that are available only when certain other parameters are used, set the value of ArgumentList to the parameters that trigger the dynamic parameters.

To detect the dynamic parameters that a provider adds to a cmdlet, set the value of ArgumentList to a path in the provider drive. When the command is a PowerShell core provider cmdlet, enter only one path in each command; the provider cmdlets return only the dynamic parameters for the first path the value of ArgumentList.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: No. | | Accept wildcard characters: No |

-CommandType <CommandTypes> — (alias Type) Specifies the kinds of commands to include. The valid values are:

* Alias: All aliases in the current session.
* All: All command types. This is the equivalent of Get-Command \*.
* Application: All non-PowerShell files in paths listed in the environment variable $env:path.
* Cmdlet: The cmdlets in the current session.
* ExternalScript: All .ps1 files in the paths listed in the environment variable $env:path.
* Filter: All functions.
* Function: All functions.

Windows PowerShell: Another valid value is Script: Script blocks in the current session.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: "Alias,Function,Filter,Cmdlet" |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-Module <object[]> — (alias PSSnapin) Specifies the modules to include, either by name or module description object (§4.5.12).

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Name <string[]> — Specifies the cmdlets or command elements to be included.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByValue and ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Noun <string[]> — Specifies cmdlets and function names that include the specified noun/noun patterns (§13).

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Syntax [ <SwitchParameter> ] — Specifies the data to be retrieved.

* For aliases, the standard name.
* For cmdlets, functions, filters, and scripts, the syntax for calling them.
* For applications, the full pathname.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-TotalCount <int> — Specifies the maximum number of commands or command elements.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-Verb <string[]> — Specifies cmdlets and function names that include the specified verb/verb patterns (§13).

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

For the Name parameter, one or more strings can be written to the pipeline.

Outputs:

When the parameter Syntax is used, a string is returned. Otherwise, a single object or an unconstrained 1-dimensional array of objects as follows, is returned:

* For each alias, an alias description object (§4.5.4)
* For each application, an application description object (§4.5.7)
* For each cmdlet, a cmdlet description object (§4.5.8)
* For each external script, an external script description object (§4.5.9)
* For each filter, a filter description object (§4.5.11)
* For each function, a function description object (§4.5.10)

Get-Command returns the commands in alphabetical order by name. When the session contains more than one command with the same name, Get-Command returns the commands in execution order.

Examples:

Get-Command -CommandType Alias -TotalCount 10  
Get-Command -Name "Get-Date" -ArgumentList "Day","Month","Year"   
Get-Command -Name "Get-Date" -CommandType Cmdlet -Syntax

## Get-Content (alias cat, gc, type)

Synopsis:

Gets the content of the items (§3.3) at the specified locations.

Syntax:

Get-Content [ -LiteralPath ] <string[]> [ -Credential <Credential> ]  
[ -Exclude <string[]> ] [ -Filter <string> ] [ -Force ] [ -Include <string[]> ]  
[ -ReadCount <long> ] [ -TotalCount <long> ] [ <CommonParameters> ]

Get-Content [ -Path ] <string[]> [ -Credential <Credential> ]  
[ -Exclude <string[]> ] [ -Filter <string> ] [ -Force ] [ -Include <string[]> ]  
[ -ReadCount <long> ] [ -TotalCount <long> ] [ <CommonParameters> ]

Description:

This cmdlet gets the content of the items at the locations specified, such as the text in a file. It reads the content one line at a time and returns an object for each line.

Parameters:

-Credential <Credential> — Specifies a user account that has permission to perform this action. The default is the current user. See §4.5.23

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-Exclude <string[]> — Specifies the path(s) to exclude from the operation.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByValue or ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Filter <string> — Specifies a file filter in the provider's format or language. This parameter qualifies the Path parameter. The syntax of the filter, including the use of wildcards, depends on the provider.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Name | Default value: None |
| Accept pipeline input: Yes, ByValue or ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Force [ <SwitchParameter> ] — Overrides restrictions that prevent the command from succeeding, just so the changes do not compromise security.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Include <string[]> — Specifies the path(s) to include in the operation.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByValue or ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-LiteralPath <string[]> — (alias PSPath) Specifies the path(s) to the item(s). The strings are used exactly as they are written. Characters that look like wildcards are not interpreted as such.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Path <string[]> — Specifies the path(s) to the item(s).

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByValue or ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-ReadCount <long> — Specifies the number of lines of content to be written to the pipeline at a time. A value of zero causes all of the content to be written together.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Name | Default value: 1 |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-TotalCount <long> — Specifies the number of lines of content to be retrieved.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Name | Default value: -1 (all lines) |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

None

Outputs:

A single object or an unconstrained 1-dimensional array of objects that describes the content retrieved. (For example, a text file or an environment variable result in output of type string, while a variable of type int has a value of that type.)

Examples:

Get-Content "J:\Test\File2.txt" -TotalCount 3  
Get-Content "Env:\Path"  
Get-Content "Variable:\Count"

## Get-Credential

Gets a credential object (§4.5.23) based on a user name and password.

Syntax:

Get-Credential [ -Credential ] <Credential> [ <CommonParameters> ]

Description:

This cmdlet creates a credential object for a specified user name and password. If the Credential parameter is omitted, the cmdlet prompts the user for a user name and password. If the Credential parameter is present, the username from within it is used and the cmdlet prompts the user for a password.

Parameters:

-Credential <Credential> — Specifies a user account that has permission to perform this action. The default is the current user. The username can be specified as a string.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

None

Outputs:

A credential object (§4.5.23).

Examples:

$v = Get-Credential  
$v = Get-Credential -Credential "User10"

## Get-Date

Synopsis:

This cmdlet gets a date-time object (§4.5.19) that represents the current or the specified date.

Syntax:

Get-Date [ -Format <string> ] [ [ -Date ] <string> ] [ -Day <int> ]  
[ -DisplayHint { Date | Time | DateTime } ] [ -Hour <int> ] [ -Minute <int> ]  
[ -Month <int> ] [ -Second <int> ] [ -Year <int> ] [ <CommonParameters> ]

Get-Date [ -UFormat <string> ] [ [ -Date ] <DateTime> ] [ -Day <int> ]  
[ -DisplayHint { Date | Time | DateTime } ] [ -Hour <int> ] [ -Minute <int> ]  
[ -Month <int> ] [ -Second <int> ] [ -Year <int> ] [ <CommonParameters> ]

Description:

This cmdlet gets a date-time object that represents the current or the specified date. Alternatively, it can get a date-time string in a number of different formats. When a date-time object is returned, any date-time component parameters that are omitted take on default values from the current date and time.

Parameters:

-Date <string> — Specifies a date and time in a format that is standard for the system locale, such as "dd‑MM‑yyyy HH:mm:ss" (German [Germany]) or "MM/dd/yyyy h:mm:ss tt" (English [United States]). By default, Get-Date gets the current system date and time.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByValue or ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-Day <int> — Specifies the day of the month in the range 1–31. If the value is greater than the number of days in the month, the given number of days is added to the first day of the month resulting in a date in the following month. For example, -Month 11 -Day 31 results in December 1, not November 31.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-DisplayHint <DisplayHintType> — Determines which elements of the date and time are written to the pipeline. The valid values are:

* Date: displays only the date
* Time: displays only the time
* DateTime: displays the date and time

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: "DateTime" |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Format <string> — Specifies the format of the date-time string written to the pipeline. For a complete definition of the format specification, see the type System.DateTimeFormatInfo in Ecma Technical Report TR/84.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Minute <int> — Specifies the minute of the hour in the range 0–59.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Hour <int> — Specifies the hour of the day in the range 0–23.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Month <int> — Specifies the month of the year in the range 1–12.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Second <int> — Specifies the second of the minute in the range 0–59.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-UFormat <string> — Specifies the UNIX-like format of the date-time string written to the pipeline.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

Here are the valid format specifiers, each of which must be preceded by %:

| **Format specifier** | **Meaning** | **Example** |
| --- | --- | --- |
| A | Day of the week - full name | Monday |
| a | Day of the week - abbreviated name | Mon |
| B | Month name - full | January |
| b | Month name - abbreviated | Jan |
| C | Century | 20 for 2006 |
| c | Date and time - abbreviated | Fri Jun 16 10:31:27 2006 |
| D | Date in mm/dd/yy format | 06/14/06 |
| d | Day of the month - 2 digits | 05 |
| e | Day of the month - digit preceded by a space | <space>5 |
| G | Same as 'Y' |  |
| g | Same as 'y' |  |
| H | Hour in 24-hour format | 17 |
| h | Same as 'b' |  |
| I | Hour in 12 hour format | 05 |
| j | Day of the year | 1-366 |
| k | Same as 'H' |  |
| l | Same as 'I' | 05 |
| M | Minutes | 35 |
| m | Month number | 06 |
| n | newline character |  |
| p | AM or PM |  |
| R | Time in 24-hour format - no seconds | 17:45 |
| r | Time in 12-hour format | 09:15:36 AM |
| S | Seconds | 05 |
| s | Seconds elapsed since January 1, 1970 00:00:00 | 1150451174.95705 |
| t | Horizontal tab character |  |
| T | Time in 24 hour format | 17:45:52 |
| U | Same as 'W' |  |
| u | Day of the week - number | Monday = 1 |
| V | Week of the year | 01-53 |
| w | Same as 'u' |  |
| W | Week of the year | 00-52 |
| X | Same as 'T' |  |
| x | Date in standard format for locale | 09/12/07 for English-US |
| Y | Year in 4-digit format | 2006 |
| y | Year in 2-digit format | 06 |
| Z | Time zone offset from Universal Time Coordinate (UTC) | -07 |

-Year <int> — Specifies the year of the minute in the range 0–9999.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

None.

Outputs:

If the Format or UFormat parameters are present, a string that represents the selected date-time; otherwise, a date-time object (§4.5.19) that represents the selected date-time.

Examples:

When run on 2010-03-15 at 12:38:01

Get-Date -Date "2010-2-1 10:12:14 pm" # 02/01/2010 22:12:14  
Get-Date -Format "m" # March 15  
Get-Date -Format "dd-MMM-yyyy" # 15-Mar-2010  
Get-Date -UFormat "%Y-%m-%d %A %Z" # 2010-03-15 Monday -04  
Get-Date -Day 2 -Month 3 -Year 2006 # 03/02/2006 12:38:01  
Get-Date -Hour 11 -Minute 3 -Second 23 # 03/15/2010 11:03:23

## Get-Help (alias help, man)

Synopsis:

Displays information about the specified command. See §A for information about creating script files that contain help comments.

Syntax:

Get- Help [ -Full ] [ [ -Name ] <string> ] [ -Category <string[]> ]  
[ -Component <string[]> ] [ -Functionality <string[]> ] [ -Online ]  
[ -Path <string> ] [ -Role <string[]> ] [ <CommonParameters> ]

Get- Help [ -Detailed ] [ [ -Name ] <string> ] [ -Category <string[]> ]  
[ -Component <string[]> ] [ -Functionality <string[]> ] [ -Online ]  
[ -Path <string> ] [ -Role <string[]> ] [ <CommonParameters> ]

Get- Help [ -Examples ] [ [ -Name ] <string> ] [ -Category <string[]> ]  
[ -Component <string[]> ] [ -Functionality <string[]> ] [ -Online ]  
[ -Path <string> ] [ -Role <string[]> ] [ <CommonParameters> ]

Get-Help [ -Parameter <string> ] [ [ -Name ] <string> ] [ -Category <string[]> ]  
[ -Component <string[]> ] [ -Functionality <string[]> ] [ -Online ]  
[ -Path <string> ] [ -Role <string[]> ] [ <CommonParameters> ]

Description:

This cmdlet displays information about the specified topic, which might be a concept or a command. The format in which this information is presented is unspecified.

To get a list of all topic titles, type Get-Help \*.

Typing Get-Help <name>, where <name> is a help topic or a word unique to a help topic, results in Get-Help displaying the topic contents. If <name> is a word or wildcard pattern that appears in several help topic titles, Get-Help displays a list of the matching titles. If the word does not appear in any help topic titles, Get-Help displays a list of topics that include that word in their contents.

The display can be configured to show an entire help file or selected parts of that file, such as the syntax, parameters, or examples.

The display items name, syntax, parameter list, parameter attribute table, common parameters, and remarks are automatically generated by Get-Help. The text for these does not derive from help comments.

Parameters:

-Category <string[]> — Displays help for items in the specified categories. Valid values are Alias, Cmdlet, Provider, and HelpFile. Category is a property of the MamlCommandHelpInfo object that Get-Help returns.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Detailed [ <SwitchParameter> ] — Adds parameter descriptions and examples to the basic help display.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Examples [ <SwitchParameter> ] — Displays only the name, synopsis, and examples.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Full [ <SwitchParameter> ] — Displays the entire help file.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Name <string> — Requests help about the topic designated by string, where string is the name of that help topic’s script file. To get help for a script file that is not located in the current path, include path information as well as the file name of the script.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-Online [ <SwitchParameter> ] — Displays the online version of a help topic in the default Internet browser. Get-Help uses the URI that appears in the first item of the Related Links section of a help topic. The help topic must include a URI that begins with "Http" or "Https" and an Internet browser must be installed on the host system.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Path <string> — Gets help that explains how the cmdlet works in the specified provider path. This parameter gets a customized version of a cmdlet help topic that explains how the cmdlet works in the specified Windows PowerShell provider path. This parameter is effective only for help about a provider cmdlet and then only when the provider includes a custom version of the provider cmdlet help topic.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

None.

Outputs:

This cmdlet writes to the pipeline an object that represents help information. The type of that object is not defined by this specification.

Examples:

Get-Help Get-\*  
Get-Help Add-Content -Examples  
Get-Help Add-Content -Full  
Get-Help E:\Scripts\Help\Get-Power

## Get-Item (alias gi)

Synopsis:

Gets one or more items from the specified locations.

Syntax:

Get-Item [ -LiteralPath ] <string[]> [ -Credential <Credential> ]  
[ -Exclude <string[]> ] [ -Filter <string> ] [ -Force ] [ -Include <string[]> ]  
[ <CommonParameters> ]

Get-Item [ -Path ] <string[]> [ -Credential <Credential> ]  
[ -Exclude <string[]> ] [ -Filter <string> ] [ -Force ] [ -Include <string[]> ]  
[ <CommonParameters> ]

Description:

This cmdlet gets the item(s) at the specified location(s).

Parameters:

-Credential <Credential> — Specifies a user account that has permission to perform this action. The default is the current user. See §4.5.23

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-Exclude <string[]> — Specifies the path(s) to exclude from the operation.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByValue or ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Filter <string> — Specifies a file filter in the provider's format or language. This parameter qualifies the Path parameter. The syntax of the filter, including the use of wildcards, depends on the provider.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Name | Default value: None |
| Accept pipeline input: Yes, ByValue or ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Force [ <SwitchParameter> ] — Allows the cmdlet to get items that cannot otherwise be accessed, such as hidden items. Implementation varies from one provider to another. However, this parameter cannot be used to override security restrictions.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Include <string[]> — Specifies the path(s) to include in the operation.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByValue or ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-LiteralPath <string[]> — (alias PSPath) Specifies the path(s) to the item(s). The strings are used exactly as they are written. Characters that look like wildcards are not interpreted as such.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Path <string[]> — Specifies the path(s) to the item(s).

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByValue or ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

A path (but not a literal path) can be written to the pipeline to Get-Item.

Outputs:

A single object (§4.5) or an unconstrained 1-dimensional array of objects that describes the item(s) retrieved.

Examples:

Get-Item -Path "J:\Test","J:\Test3"  
Get-Item Env:Day1,Env:Day2  
Get-Item -Path "Function:MyFun2"  
Get-Item -Path "Variable:MyVar1"

## Get-Location (alias gl, pwd)

Synopsis:

Gets information about the current working location (§3.1.4) for the specified drive(s), or the working locations for the specified stack(s).

Syntax:

Get-Location [ -PSDrive <string[]> ] [ -PSProvider *<string[]>* ] [ CommonParameters> ]

Get-Location [ -Stack ] [ -StackName *<string[]>* ] [ <CommonParameters> ]

Description:

For each drive specified, this cmdlet creates an object that represents the current working location. For each stack specified, this cmdlet creates an object that represents all the working locations on that stack.

Parameters:

-PSDrive <string[]> — Specifies one or more drives. If this parameter is omitted, the current drive is used.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-PSProvider <string[]> — Gets the current location in the drive(s) supported by the specified provider. If the specified provider supports more than one drive, Get-Location returns the location on the most recently accessed drive.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Stack [ <SwitchParameter> ] — Specifies that the current working location stack be used instead of one or more drives. If both Stack and StackName are present, Stack is ignored.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-StackName <string[]> — Specifies that the named working location stack(s) be used. If both Stack and StackName are present, Stack is ignored. $null and "" both indicate the current working location stack. "default" indicates the default working location stack at session startup.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

None.

Outputs:

If a single drive is specified or implied, a single current working location object (§4.5.5) is output; otherwise, an unconstrained 1-dimensional array of such objects is output with each element being the current working location object for the corresponding drive, in order.

If a single stack is specified (by Stack or StackName), a single stack of working location objects (§4.5.5) is output; otherwise, an unconstrained 1-dimensional array of such objects is output with each element being the working location objects for the corresponding stack, in order.

Examples:

Get-Location # get location of current drive  
Get-Location -PSDrive G,C,D # get location of specified drives  
Get-Location -Stack # get all locations from current stack  
Get-Location -StackName "Stack2","Stack1"  
 # get all locations from named stacks

## Get-Member (alias gm)

Synopsis:

Gets the specified members of an object.

Syntax:

Get-Member [ [ -Name ] <string[]> ] [ -Force ] [ -InputObject <object> ]  
[ -MemberType ] { AliasProperty | NoteProperty | ScriptProperty  
| PropertySet | ScriptMethod | MemberSet } [ -Static ]  
[ -View { Extended | Adapted | Base | All ] [ <CommonParameters>]

Description:

This cmdlet gets information about the properties and methods of an object.

Windows PowerShell: The following extra member types are permitted: CodeProperty and CodeMethod.

Parameters:

-Name <string[]> — Specifies the names of the members to be retrieved. If the Name parameter is used in conjunction with the MemberType, View, or Static parameters, only the members that satisfy the criteria of all parameters are retrieved.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Position 1 | Default value: All instance members |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Force [ <SwitchParameter> ] — This switch parameter causes the following intrinsic members to be included in the output:

* PSBase: The base properties of the object without extension or adaptation (§4.6).
* PSAdapted: The members defined in the PowerShell Extended Type System (ETS) (§4.6).
* PSExtended: The members that were added in the Types.ps1xml files or by using the Add-Member cmdlet (§4.6, §13.2).
* PSTypeNames: A list of object types that describe the object, in order of specificity. When formatting the object, PowerShell searches for the types in the Format.ps1xml files (§4.6) in the PowerShell installation directory ($PSHome). It uses the formatting definition for the first type that it finds.

Windows PowerShell: The intrinsic member PsObject is also included. This member provides access to a member set that only allows access to the PsObject instance wrapping the object. The wrapped object may or may not already be a PsObject.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-InputObject <object> — The object to from which the members are to be retrieved. Note that the use of this parameter is not the same as piping an object to Get-Member. Specifically, when a collection of objects is piped to Get-Member, that cmdlet gets the members of the individual objects in the collection, such as the properties of the integers in an array of integers. However, when InputObject is used to submit a collection of objects, Get-Member gets the members of the collection, such as the properties of the array in an array of integers.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByValue (§12.3.7) | | Accept wildcard characters: No |

-MemberType <PsMemberTypes> —Specifies the member type of the members to be retrieved. By default, all of them are retrieved. The valid values are:

* AliasProperty: A new name for an existing property.
* MemberSet: A predefined collection of properties and methods, such as PsBase and PsTypeNames (§4.6).
* NoteProperty: A property with a static value.
* PropertySet: A predefined collection of object properties.
* ScriptMethod: A method whose value is the output of a script.
* ScriptProperty: A property whose value is the output of a script.

If the Name parameter is used in conjunction with the MemberType, View, or Static parameters, only the members that satisfy the criteria of all parameters are retrieved.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: none |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Static [ <SwitchParameter> ] — Specifies that only the static members are to be retrieved. If both the Static parameter and the View parameter are used, the View parameter is ignored. If the Name parameter is used in conjunction with the MemberType or Static parameters, only the members that satisfy the criteria of all parameters are retrieved.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-View <PsMemberViewTypes> —Specifies the categories of members to be retrieved. The valid values are:

* Base: Gets only the base members of the object without extension or adaptation (§4.6).
* Adapted: Gets only the members defined in the PowerShell Extended Type System (ETS) (§4.6).
* Extended: Gets only the members that were added in the Types.ps1xml files or by using the Add-Member cmdlet (§4.6, §13.2).
* All: Gets the members in the Base, Adapted, and Extended views.

If both the Static parameter and the View parameter are used, the View parameter is ignored. If the Name parameter is used in conjunction with the MemberType, View, or Static parameters, only the members that satisfy the criteria of all parameters are retrieved.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: "Adapted,Extended" |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

Any object can be piped to this cmdlet.

Outputs:

A single object (§4.5.25) or an unconstrained 1-dimensional array of objects that describes the member(s) retrieved.

Examples:

$v = New-Item -Force -Path "I:\" -Name "Test" -Itemtype "Directory"  
$v | Get-Member  
$v | Get-Member -Force  
$v | Get-Member -Name Name,Extension,CreationTime  
$v | Get-Member -MemberType NoteProperty,ScriptProperty  
$v | Get-Member -View All

## Get-Module (alias gmo)

Synopsis:

Gets an object for each module (§3.14) that has been imported or that can be imported into the current session.

Syntax:

Get-Module [ [ -Name ] <string[]> ] [ -All ] [ -ListAvailable ]  
[ <CommonParameters> ]

Description:

This cmdlet gets objects for each module that has been imported into the current session, or that can be imported.

Parameters:

-All [ <SwitchParameter> ] — Gets module objects for all module files. In the absence of this parameter, Get-Module gets the module object for the default module file only. The cmdlet selects file types in the following order: manifest files, script module files, and binary module files.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-ListAvailable [ <SwitchParameter> ] — Gets all of the modules that can be imported into the session. Get-Module gets the modules in the paths specified by the environment variable PSModulePath. Without this parameter, Get-Module gets only the modules that have been imported into the session.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Name <string[]> — Gets the specified modules.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByValue (§12.3.7) | | Accept wildcard characters: Yes |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

Module names can be written to the pipeline to Get-Module.

Outputs:

A single object (§4.5.12) or an unconstrained 1-dimensional array of objects that describes the module(s) retrieved.

Examples:

Get-Module -All -ListAvailable  
Get-Module PSTest\_Temperature

## Get-PSDrive (alias gdr)

Synopsis:

Gets the PowerShell drives (§3.1) in the current session.

Syntax:

Get-PSDrive [ -LiteralName ] <string[]> [ -PSProvider <string[]> ]  
[ -Scope <string> ] [ <CommonParameters> ]

Get-PSDrive [ -Name ] <string[]> [ -PSProvider <string[]> ]  
[ -Scope <string> ] [ <CommonParameters> ]

Description:

This cmdlet gets the one or more PowerShell drives in the current session.

Parameters:

-LiteralName <string[]> — Specifies the name of the PowerShell drives (without trailing colon).

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-Name <string[]> — Specifies the name of the PowerShell drive (without trailing colon).

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-PSProvider <string[]> — Specifies the names of the providers whose information is to be retrieved.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Scope <string> — Gets the PowerShell drives in the specified scope. Valid values are "Global", "Local", and "Script", or a number.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: "Local" |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

None.

Outputs:

If only one drive is selected, an object (§4.5.2) describing that drive is output to the pipeline; otherwise, an unconstrained 1-dimensional array of such objects is output.

Examples:

Get-PSDrive  
Get-PSDrive D  
Get-PSDrive -PSProvider filesystem

## Get-PSProvider

Synopsis:

Gets information about one or more providers (§3.1).

Syntax:

Get-PSProvider [ [ -PSProvider ] <string[]> ] [ <CommonParameters> ]

Description:

This cmdlet gets information about one or more providers in the current session.

Parameters:

-PSProvider <string[]> — Specifies the names of the providers whose information is to be retrieved.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

None.

Outputs:

If only one provider is selected, an object (§4.5.1) describing that provider is output to the pipeline; otherwise, an unconstrained 1-dimensional array of such objects is output.

Examples:

Get-PSProvider # request information for all providers  
Get-PSProvider "Alias","Variable" # request information for 2 providers  
  
$v = Get-PSProvider "FileSystem" # request information for 1 provider  
foreach ($e in $v.Drives) { … } # process each drive

## Get-Variable (alias gv)

Synopsis:

Writes information about the specified variables to the pipeline.

Syntax:

Get-Variable [ -Name ] <string[]> [ -Exclude *<string[]>* ] [ -Include *<string[]>* ]  
[ -Scope <string> ] [ -ValueOnly ] [ <CommonParameters> ]

Description:

This cmdlet writes information about the specified variables to the pipeline. The amount of information can be limited by the ValueOnly parameter.

Parameters:

-Exclude <string[]> — Specifies the name(s) to be excluded from the operation.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: Yes |

-Include <string[]> — Specifies the name(s) to be included in the operation.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: Yes |

-Name <string[]> — Specifies the name(s) of the variables whose information is to be written.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Scope <string> — Specifies the scope (§3.5) of the variable whose information is to be written. The valid values are "Global", "Script", and "Local", or scope number (§3.5.2).

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-ValueOnly [ <SwitchParameter> ] — Specifies that only the variable values are to be written rather than their full information.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

A string containing a variable name can be written to the pipeline to Get-Variable.

Outputs:

One object (§4.5.3) per variable.

Examples:

Get-Variable 'Count10?' -Exclude 'Count101','Count102' -Scope 'Script'

## Group-Object (alias group)

Synopsis:

Groups objects that contain the same value for specified properties.

Syntax:

Group-Object [ -AsHashTable ] [ -AsString ] [ [ -Property ] <object[]> ]  
[ -CaseSensitive ] [ -Culture <string> ] [ -InputObject <object> ]  
[ -NoElement ] [ CommonParameters> ]

Description:

This cmdlet groups objects based on the value of a specified property. It returns a table with one row for each property value and a column that displays the number of items with that value. If more than one property is specified, the items are grouped by the values of the first property, and then within each property group, by the value of the next property, and so on.

Parameters:

-AsHashTable [ <SwitchParameter> ] — (alias AHT) Returns the group as a hash table. The keys of the hash table are the property values by which the objects are grouped. The values of the hash table are the objects that have that property value. By itself, this parameter returns each hash table in which each key is an instance of the grouped object. When used with the AsString parameter, the keys in the hash table are strings.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: none |
| Accept pipeline input: No | | Accept wildcard characters: No |

-AsString [ <SwitchParameter> ] — Converts the hash table keys to strings. By default, the hash table keys are instances of the grouped object. This parameter is valid only when used with the AsHashTable parameter.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: none |
| Accept pipeline input: No | | Accept wildcard characters: No |

-CaseSensitive [ <SwitchParameter> ] — Makes the grouping case-sensitive. Without this parameter, the property values of objects in a group might have different cases.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Culture [ <string> ] — Specifies the culture to use when comparing strings (as in "en-US" for US English using language codes from ISO 639-1 and country codes from ISO 3166-1).

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: Yes |

-InputObject <object> — Specifies the objects to group, as a collection. As a result, this cmdlet creates a single group with that object as its member.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByValue (§12.3.7) | | Accept wildcard characters: No |

-NoElement [ <SwitchParameter> ] — Omits the members of a group from the results.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Property <object[]> — Specifies the properties for grouping. The objects are arranged into groups based on the value of the specified property, which can be a property calculated using a hashtable.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

Any object type can be piped to this cmdlet.

Outputs:

When AsHashTable is present, a hash table is output; otherwise, a single object or an unconstrained 1-dimensional array of group-info objects (§4.5.20) is output.

Examples:

Get-ChildItem \*.xml | Group-Object -Property Length -AsHashTable  
Get-ChildItem \*.\* | Sort-Object -Property Extension | Group-Object  
 -Property Extension

Windows PowerShell: Group-Object does not require that the objects being grouped be of the same Microsoft .NET Framework type. When grouping objects of different .NET Framework types, Group-Object uses the following rules:

-- Same Property Names and Types: If the objects have a property with the specified name, and the property values have the same .NET Framework type, the property values are grouped by using the same rules that would be used for objects of the same type.

-- Same Property Names, Different Types: If the objects have a property with the specified name, but the property values have a different .NET Framework type in different objects, Group-Object uses the .NET Framework type of the first occurrence of the property as the .NET Framework type for that property group. When an object has a property with a different type, the property value is converted to the type for that group. If the type conversion fails, the object is not included in the group.

-- Missing Properties: Objects that do not have a specified property are considered ungroupable. Ungroupable objects appear in the final GroupInfo object output in a group named AutomationNull.Value.

## Import-Module (alias ipmo)

Synopsis:

Adds one or more modules (§3.14) to the current session.

Syntax:

Import-Module [ -Name ] <string[]> [ -Alias <string[]> ]  
[ -ArgumentList <object[]> ] [ -AsCustomObject ] [ -Cmdlet <string[]> ]  
[ -DisableNameChecking ] [ -Force ] [ -Function <string[]> ] [ -Global ]  
[ -PassThru ] [ -Prefix <string> ] [ -Variable <string[]> ]  
[ -Version <Version> ] [ <CommonParameters> ]

Import-Module [ -Assembly ] <Assembly[]> [ -Alias <string[]> ]  
[ -ArgumentList <object[]> ] [ -AsCustomObject ] [ -Cmdlet <string[]> ]  
[ -DisableNameChecking ] [ -Force ] [ -Function <string[]> ] [ -Global ]  
[ -PassThru ] [ -Prefix <string> ] [ -Variable <string[]> ]  
[ -Version <Version> ] [ <CommonParameters> ]

Import-Module [ -ModuleInfo ] <PSModuleInfo[]> [ -Alias <string[]> ]  
[ -ArgumentList <object[]> ] [ -AsCustomObject ] [ -Cmdlet <string[]> ]  
[ -DisableNameChecking ] [ -Force ] [ -Function <string[]> ] [ -Global ]  
[ -PassThru ] [ -Prefix <string> ] [ -Variable <string[]> ]  
[ -Version <Version> ] [ <CommonParameters> ]

Description:

This cmdlet imports one or more modules to the current session. By default, Import-Module imports all members that the module exports.

Parameters:

-Alias <string[]> — Imports only the specified aliases from the module into the current session. Some modules automatically export selected aliases into the session when the module is imported. This parameter allows selection from among the exported aliases.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: Yes |

-ArgumentList <object[]> — (alias Args) Specifies arguments to be passed to a script module during the Import-Module command. This parameter is valid only when importing a script module.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-AsCustomObject [ <SwitchParameter> ] — Returns a custom object with members that represent the imported module members. This parameter is valid for script modules only. (Exported functions turn into script methods, and exported variables turn into note properties. By default, only functions are turned into script methods, but Export-ModuleMember (§13.11) can specify exactly what gets exported.)

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Assembly <Assembly[]> — Imports the cmdlets and providers implemented in the specified assembly objects. When this parameter is used, only the cmdlets and providers implemented by the specified assemblies are imported. If the module contains other files, they are not imported. Use this parameter for debugging and testing the module, or when instructed to use it by the module author.

|  |  |  |
| --- | --- | --- |
| Required: True | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByValue (§12.3.7) | | Accept wildcard characters: Yes |

-Cmdlet <string[]> — Imports only the specified cmdlets from the module into the current session. Some modules automatically export selected cmdlets into a session when the module is imported. This parameter allows selection from among the exported cmdlets.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: Yes |

-DisableNameChecking [ <SwitchParameter> ] — Suppresses the message that warns when a cmdlet or function whose name includes an unapproved verb or a prohibited character, is imported.

By default, when an imported module exports cmdlets or functions that have unapproved verbs in their names, the following warning message is issued: "WARNING: Some imported command names include unapproved verbs which might make them less discoverable. Use the Verbose parameter for more detail …" This message is only a warning. The complete module is still imported, including the non-conforming commands. Although the message is displayed to module users, the naming problem should be fixed by the module author.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Force [ <SwitchParameter> ] — Re-imports a module and its members, even if the module or its members have an access mode of read-only.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Function <string[]> — Imports only the specified functions from the module into the current session. Enter a list of functions. Some modules automatically export selected functions into the session when the module is imported. This parameter allows selection from among the exported functions.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: Yes |

-Global [ <SwitchParameter> ] — When used in a script module, this parameter imports modules into the global session state. Otherwise, it is ignored. By default, the commands in a script module, including commands from nested modules, are imported into the caller's session state.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-ModuleInfo <PSModuleInfo[]> — Specifies the module objects to import.

|  |  |  |
| --- | --- | --- |
| Required: True | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByValue (§12.3.7) | | Accept wildcard characters: No |

-Name <string[]> — Specifies the names of the modules to import. Use the name of the module or the name of a file in the module. File paths are optional. If the path is omitted, Import-Module looks for the module in the paths saved in PSModulePath. Whenever possible, specify the module name only. If a file name is used, only the members implemented in that file are imported. If the module contains other files, they are not imported.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByValue (§12.3.7) | | Accept wildcard characters: Yes |

-PassThru [ <SwitchParameter> ] — Returns objects that represent the modules that were imported. By default, this cmdlet does not generate any output. (When the output of a Get-Module -ListAvailable command is piped to an Import-Module command with the PassThru parameter, Import-Module returns the object that Get-Module passed to it without updating the object. As a result, the Exported and NestedModules properties are not yet populated. When the Prefix parameter is used to specify a prefix for the member, the prefix does not appear in the member names in the properties of the module object. The object records what was exported before the prefix was applied.)

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Prefix <string> — Adds the specified prefix to the nouns in the names of imported module members. Use this parameter to avoid name conflicts that might occur when different members in the session have the same name. This parameter does not change the module, and it does not affect files that the module imports for its own use (i.e., nested modules). It affects only the names of members in the current session.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: Yes |

-Variable <string[]> — Imports only the specified variables from the module into the current session. Some modules automatically export selected variables into the session when the module is imported. This parameter allows selection from among the exported variables.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: Yes |

-Version <Version> — Specifies the version of the module to import. Use this parameter when there are different versions of the same module on the host system.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

Module names, module objects, or assemblies can be written in the pipeline to Import-Module.

Outputs:

By default, Import-Module does not generate any output. However, if the PassThru parameter is present, a Module Information object (§4.5.12) that represents the module is output. If the AsCustomObject parameter is present, a Custom information object (§4.5.13) is output.

Examples:

Import-Module -Name Lib1,Lib2 -Verbose  
Import-Module "E:\Modules\PSTest\_Temperature"

## Invoke-Item (alias ii)

Synopsis:

Perform the default action on the specified item(s) (§3.3).

Syntax:

Invoke-Item [ -LiteralPath ] <string[]> [ -Credential <Credential> ]  
[ -Exclude <string[]> ] [ -Filter <string> ] [ -Include <string[]> ] [ -Confirm ]  
[ -WhatIf ] [ <CommonParameters> ]

Invoke-Item [ -Path ] <string[]> [ -Credential <Credential> ]  
[ -Exclude <string[]> ] [ -Filter <string> ] [ -Include <string[]> ] [ -Confirm ]  
[ -WhatIf ] [ <CommonParameters> ]

Description:

This cmdlet performs the default action on the specified item(s). For example, it runs an executable file or opens a document file in the application associated with the document file type. The default action depends on the type of item and is determined by the provider. Once a default action has been performed, the cmdlet continues execution; it does not wait for the default action to complete.

Parameters:

-Confirm [ <SwitchParameter> ] — (alias cf) Prompts for confirmation before executing the cmdlet. If confirmation is denied, the cmdlet terminates with the new variable being created or changed.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Credential <Credential> — Specifies a user account that has permission to perform this action. The default is the current user. See §4.5.23

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-ExcludePath <string[]> — Specifies the path(s) to exclude from the operation.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Filter <string> — Specifies a file filter in the provider's format or language. This parameter qualifies the Path parameter. The syntax of the filter, including the use of wildcards, depends on the provider.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-IncludePath <string[]> — Specifies the path(s) to include in the operation.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-LiteralPath <string[]> — (alias PSPath) Specifies the path(s) to the item(s) on which the default action is to be performed. The strings are used exactly as they are written. Characters that look like wildcards are not interpreted as such.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Path <string[]> — Specifies the path(s) to the item(s) on which the default action is to be performed.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByValue or ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-WhatIf [ <SwitchParameter> ] — (alias wi) Describes what would happen if the cmdlet were executed without actually executing it.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

A path (but not literal path) can be written to the pipeline to Remove-Item.

Outputs:

None by the cmdlet; however, output might be generated by the items that are invoked.

Examples:

Invoke-Item File2.txt  
Invoke-Item "J:\Manual.pdf"  
Invoke-Item "J:\Capture.jpg","J:\Action list.doc"

## Join-Path

Synopsis:

Combines a path (§3.4) and a child path into a single path.

Syntax:

Join-Path [ -Path ] <string[]> [ -ChildPath ] <string[]>  
[ -Credential <Credential> ] [ -Resolve ] [ <CommonParameters> ]

Description:

This cmdlet combines a path and child-path into a single path. The provider supplies the path delimiters. It can also retrieve all the items have the joined pathname.

Parameters:

-ChildPath <string> — Specifies the elements to append to the value(s) of Path.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 2 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Credential <Credential> — Specifies a user account that has permission to perform this action. The default is the current user. See §4.5.23

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-Path <string[]> — Specifies the main path (or paths) to which the child-path is appended. The value of Path determines which provider joins the paths and adds the path delimiters.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByName or ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Resolve [ <SwitchParameter> ] — Retrieves the items referenced by the joined path.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

A string containing a path (but not a child path) can be written to the pipeline to Split-Path.

Outputs:

If the result is one path, that string is output. Otherwise, an unconstrained 1-dimensional array of string values is output, whose elements correspond to the path results.

Examples:

Join-Path -Path c:,e:\Main -ChildPath File1.txt  
Join-Path -Path G:\ -ChildPath Temp??.txt  
Join-Path -Path G:\ -ChildPath Temp\Data\*.\* -Resolve

## Measure-Object (alias measure)

Synopsis:

Calculates the numeric properties of objects, and the characters, words, and lines in string objects, such as files of text.

Syntax:

Measure-Object [ -Average ] [ -Maximum ] [ -Minimum ] [ -Sum ]  
[ [ -Property ] <string[]> ] [ -InputObject <object> ] [ CommonParameters> ]

Measure-Object [ -Character ] [ -IgnoreWhiteSpace ] [ -Line ] [ -Word ]  
[ [ -Property ] <string[]> ] [ -InputObject <object> ] [ CommonParameters> ]

Description:

This cmdlet calculates the property values of certain types of object. It performs calculations on the property values of objects; counts objects and calculates the minimum, maximum, sum, and average of the numeric values; and for text objects, it counts and calculates the number of lines, words, and characters.

Parameters:

-Average [ <SwitchParameter> ] — Displays the average value of the specified properties.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: none |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Character [ <SwitchParameter> ] — Counts the number of characters in the input object.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: none |
| Accept pipeline input: No | | Accept wildcard characters: No |

-IgnoreWhiteSpace [ <SwitchParameter> ] — Ignores white space in word counts and character counts. By default, white space is not ignored.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: none |
| Accept pipeline input: No | | Accept wildcard characters: No |

-InputObject <object> — Specifies the objects to be measured.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByValue (§12.3.7) | | Accept wildcard characters: No |

-Line [ <SwitchParameter> ] — Counts the number of lines in the input object.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: none |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Maximum [ <SwitchParameter> ] — Displays the maximum value of the specified properties.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: none |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Minimum [ <SwitchParameter> ] — Displays the minimum value of the specified properties.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: none |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Property <string[]> — Specifies one or more numeric properties to measure.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Position 1 | Default value: the Count (Length) property |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Sum [ <SwitchParameter> ] — Displays the sum of the values of the specified properties.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: none |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Word [ <SwitchParameter> ] — Counts the number of words in the input object.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: none |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

Any object type can be piped to this cmdlet.

Outputs:

When Average, Maximum, Minimum, or Sum are present, a generic-measure-info object (§4.5.21) is output. When Character, IgnoreWhiteSpace, Line, or Word are present, a text-measure-info object (§4.5.22) is output.

Examples:

Get-ChildItem \*.\* | Measure-Object -Property Length -Maximum -Minimum  
 -Average -Sum  
Get-ChildItem Test.txt | Measure-Object -Character -Line -Word

## Move-Item (alias mi, move, mv)

Synopsis:

Moves one or more items (§3.3) from one location to another.

Syntax:

Move-Item [ -LiteralPath ] <string[]> [ [ -Destination ] <string> ]  
[ -Credential <Credential> ] [ -Exclude <string[]> ] [ -Filter <string> ]  
[ -Force ] [ -Include <string[]> ] [ -PassThru ] [ -Confirm ] [ -WhatIf ]  
[ <CommonParameters> ]

Move-Item [ -Path ] <string[]> [ [ -Destination ] <string> ]  
[ -Credential <Credential> ] [ -Exclude <string[]> ] [ -Filter <string> ]  
[ -Force ] [ -Include <string[]> ] [ -PassThru ] [ -Confirm ] [ -WhatIf ]  
[ <CommonParameters> ]

Description:

This cmdlet moves one more items (including their properties, contents, and child items) from one location to another location. The locations must be supported by the same provider. Moving an item involves adding it to the new location and deleting it from its original location.

Parameters:

-Confirm [ <SwitchParameter> ] — (alias cf) Prompts for confirmation before executing the cmdlet. If confirmation is denied, the cmdlet terminates with the new variable being created or changed.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Credential <Credential> — Specifies a user account that has permission to perform this action. The default is the current user. See §4.5.23

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-Destination <string> — Specifies the destination path of the move.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 2 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-ExcludePath <string[]> — Specifies the path(s) to exclude from the operation.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Filter <string> — Specifies a file filter in the provider's format or language. This parameter qualifies the Path parameter. The syntax of the filter, including the use of wildcards, depends on the provider.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Force [ <SwitchParameter> ] — Allows the cmdlet to move an item that writes over an existing read-only item. Implementation varies from provider to provider.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-IncludePath <string[]> — Specifies the path(s) to include in the operation.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-LiteralPath <string[]> — (alias PSPath) Specifies the path(s) to the item(s) to be moved. The strings are used exactly as they are written. Characters that look like wildcards are not interpreted as such.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-PassThru [ <SwitchParameter> ] — Returns one or more objects representing the moved items. By default, this cmdlet does not generate any output.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Path <string[]> — Specifies the path(s) to the item(s) to be moved.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByValue or ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-WhatIf [ <SwitchParameter> ] — (alias wi) Describes what would happen if the cmdlet were executed without actually executing it.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

A path (but not literal path) can be written to the pipeline to Move-Item.

Outputs:

If PassThru is present, a single object (§4.5) or an unconstrained 1-dimensional array of objects that describes the item(s) moved.

Examples:

Move -Item -Path "J:\Test\\*","J:\Test3\\*" -Destination "J:\"  
Move -Item -Path "E:\Temp\\*" -Destination "J:\Test"

## New-Alias (alias nal)

Synopsis:

Creates a new alias.

Syntax:

New-Alias [ -Name ] <string> [ [ -Value ] <string> ] [ -Description <string> ]  
[ -Force ] [ -Option { None | ReadOnly | Constant | Private | AllScope } ]  
[ -PassThru ] [ -Scope <string> ] [ -Confirm ] [ -WhatIf ] [ <CommonParameters> ]

Description:

This cmdlet creates a new alias for a cmdlet or command element, and allows various characteristics of that alias to be specified.

Parameters:

-Confirm [ <SwitchParameter> ] — Prompts for confirmation before executing the cmdlet. If confirmation is denied, the cmdlet terminates with the new variable being created or changed.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Description <string> — Specifies a description of the alias.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Force [ <SwitchParameter> ] — This switch parameter allows an existing alias to be reassigned and/or its characteristics to be changed.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Name <string> — Specifies the name of the new alias. (An alias is spelled like a command.)

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-Option <ScopedItemOptions> — Sets the value of the Options property of the new alias. The valid values are:

* None: Sets no options.
* ReadOnly: The alias cannot be reassigned and its characteristics cannot be changed except by using the Force parameter.
* Constant: The alias cannot be deleted, and its characteristics cannot be changed.
* Private: The alias is available only within the scope in which it is defined, and not in any child scopes.
* AllScope: The alias is copied to any new scopes that are created, but is not put into any existing scopes.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: "None" |
| Accept pipeline input: No | | Accept wildcard characters: No |

-PassThru [ <SwitchParameter> ] — Causes the cmdlet to write to the pipeline an object that represents the new alias (§4.5.4). (Ordinarily, the cmdlet does not write any output to the pipeline.)

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Scope <string> — Specifies the scope (§3.5) of the new alias. The valid values are "Global", "Script", and "Local", or scope number (§3.5.2).

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: "Local" |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Value <string> — Specifies the name of the cmdlet or command element that is being aliased.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 2 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-WhatIf [ <SwitchParameter> ] — (alias wi) Describes what would happen if the cmdlet were executed without actually executing it.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

None.

Outputs:

None unless the PassThru switch parameter is used.

Examples:

New-Alias 'Func1' 'F1'  
New-Alias 'Func2' 'Func1'  
New-Alias 'Script1' 'E:\Powershell\Scripts\script1.ps1'  
New-Alias 'Script1' 'E:\Powershell\Scripts\script2.ps1' -Force  
New-Alias 'Func3' 'F1' -Scope Global  
  
function F1 { … }

## New-Item (alias ni)

Synopsis:

Creates a new item with the specified name and type in the specified path.

Syntax:

New-Item [ -Path ] <string[]> [ -Credential <Credential> ] [ -Force ]  
[ -ItemType <string> ] [ -Value <object> ] [ -Confirm ] [ -WhatIf ]  
[ <CommonParameters> ]

New-Item [ [ -Path ] <string[]> ] -Name <string> [ -Credential <Credential> ]  
[ -Force ] [ -ItemType <string> ] [ -Value <object> ] [ -Confirm ] [ -WhatIf ]  
[ <CommonParameters> ]

Description:

This cmdlet creates a new item with the specified name and type in the specified path, and optionally sets its value. The types of items that can be created depend upon the path of the item (see §3.1). For example, in the file system, New-Item can be used to create directories and files.

When creating an alias, the capabilities of New-Item are a subset of those provided by New-Alias (§13.33). When creating a variable, the capabilities of New-Item are a subset of those provided by New-Variable (§13.37).

Parameters:

-Confirm [ <SwitchParameter> ] — (alias cf) Prompts for confirmation before executing the cmdlet. If confirmation is denied, the cmdlet terminates with the new variable being created or changed.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Credential <Credential> — Specifies a user account that has permission to perform this action. The default is the current user. See §4.5.23

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-Force [ <SwitchParameter> ] — This switch parameter allows the cmdlet to write over an existing item. However, this parameter cannot be used to override security restrictions.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-ItemType <string> — (alias Type) Specifies the kind of new item. For the FileSystem provider the choices are "File" and "Directory".

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-Name <string> — Specifies the name of the new item. Alternatively, the name can be included in the Path parameter.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-Path <string[]> — Specifies the path(s) to the new item. The name of the new item can be included in the path or specified via the Name parameter. If multiple paths are specified, the item is created in each path.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Value <object> — Specifies the value of the new item. For a function, the value is a script block that implements the function.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByValue or ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-WhatIf [ <SwitchParameter> ] — (alias wi) Describes what would happen if the cmdlet were executed without actually executing it.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

Any kind of object for the Value parameter can be written to the pipeline to New-Item.

Outputs:

A single object (§4.5) or an unconstrained 1-dimensional array of objects that describes the item(s) created. For the FileSystem directory object description type, see §4.5.17. For the FileSystem file object description type, see §4.5.18.

Examples:

New-Item -Path "J:\" -Name "Test" -Itemtype "Directory"  
New-Item -Path "J:\Test2" -Itemtype "Directory"  
New-Item -Path "J:\Test" -Name "File1.txt" -Itemtype "File"  
New-Item -Path "J:\Test" -Name "File2.txt" -Itemtype "File" -Value "Hello`n"  
New-Item -Path "J:\Test","J:\Test3" -Name "File3.txt" -Itemtype "File"

New-Item -Path "Alias:" -Name "MyName1" -Value "F1"  
New-Item -Path "Env:" -Name "MyEnv1" -Value "Max=200"  
New-Item -Path "Function:" -Name "MyFun2" -Value { param ($p1,$p2) $p1\*$p2 }  
New-Item -Path "Variable:" -Name "MyVar1" -Value 100

## New-Module (alias nmo)

Synopsis:

Creates a dynamic module (§11.7).

Syntax:

New-Module [ -Name ] <string> [ -ScriptBlock ] <scriptblock>  
[ -ArgumentList <object[]> ] [ -AsCustomObject ] [ -Cmdlet <string[]> ]  
[ -Function <string[]> ] [ -ReturnResult ] [ <CommonParameters> ]

New-Module [ -ScriptBlock ] <scriptblock> [ -ArgumentList <object[]> ]  
[ -AsCustomObject ] [ -Cmdlet <string[]> ] [ -Function <string[]> ]  
[ -ReturnResult ] [ <CommonParameters> ]

Description:

This cmdlet creates a dynamic module from a script block. The members of the module are implemented as script methods of a custom object instead of being imported into the session.

Parameters:

-ArgumentList [ <object[]> ] — (alias Args) Specifies the arguments (if any) to be passed to the script block designated by ScriptBlock.

|  |  |  |
| --- | --- | --- |
| Required: False | Position/Named: Named | Default value: none |
| Accept pipeline input: No | | Accept wildcard characters: No |

-AsCustomObject [ <SwitchParameter> ] — Causes a custom object to be returned that represents the dynamic module. The module members are implemented as script methods of the custom object, but they are not imported into the session. This custom object can be saved in a variable and its members can be invoked using dot notation. If the module has multiple members with the same name, only one member with each name is accessible from the custom object. (Exported functions turn into script methods, and exported variables turn into note properties. By default, only functions are turned into script methods, but Export-ModuleMember (§13.11) can specify exactly what gets exported.)

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: none |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Cmdlet [ <string[]> ] — Exports only the specified cmdlets from the module into the current session. By default, all cmdlets in the module are exported. Although cmdlets cannot be defined in a script block, a dynamic module can include cmdlets if it imports them from a binary module.

|  |  |  |
| --- | --- | --- |
| Required: False | Position/Named: Named | Default value: none |
| Accept pipeline input: No | | Accept wildcard characters: Yes |

-Function [ <string[]> ] — Exports only the specified functions from the module into the current session. By default, all functions defined in a module are exported.

|  |  |  |
| --- | --- | --- |
| Required: False | Position/Named: Named | Default value: none |
| Accept pipeline input: No | | Accept wildcard characters: True |

-Name <string> — Specifies a name for the new module. The default value is implementation defined.

|  |  |  |
| --- | --- | --- |
| Required: True | Position/Named: Position 1 | Default value: implementation defined |
| Accept pipeline input: Yes, ByValue (§12.3.7) | | Accept wildcard characters: No |

-ReturnResult [ <SwitchParameter> ] — Runs the script block and returns the script block results instead of returning a module object.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: none |
| Accept pipeline input: No | | Accept wildcard characters: No |

-ScriptBlock [ <scriptblock> ] — Specifies the contents of the dynamic module.

|  |  |  |
| --- | --- | --- |
| Required: True | Position/Named: Position 1 | Default value: none |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

Module names can be written to the pipeline to New-Module.

Outputs:

By default, a module description object (§4.5.12) is output; however, if the AsCustomObject parameter is used, a Custom information object (§4.5.13); otherwise, if the ReturnResult parameter is used, the output from the script block is the output.

Examples:

New-Module -Name DynMod1 -ScriptBlock {function F1 { … } }  
New-Module -Name DynMod2 -ArgumentList 123,"abc" -ScriptBlock   
 { param ($p1,$p2) function F2 { … } }  
New-Module -Name DynMod3 -AsCustomObject -ScriptBlock {function F3 { … } }  
New-Module -Name DynMod4 -ReturnResult -ScriptBlock {function F4 { … } … }

Windows PowerShell: If Name is omitted, a name is an automatically generated beginning with "\_\_DynamicModule\_" followed by a GUID that specifies the path to the dynamic module.

## New-Object

Synopsis:

Creates an object of the given type.

Syntax:

New-Object [ -TypeName ] <string> [ [ -ArgumentList ] <object[]> ]  
[ -Property <hashtable> ] [ <CommonParameters> ]

Windows PowerShell: The second form of calling this cmdlet creates an instance of a COM object, where <string> is the ProgID.

New-Object -ComObject <string> [ -Strict ]  
[ -Property <hashtable> ] [ <CommonParameters> ]

Description:

This cmdlet creates an instance of the specified type.

| **TypeName** | **Default Initial Value** | **Permitted ArgumentList Values** |
| --- | --- | --- |
| Any value type | 0 cast to that type | None |
| object | Empty object | None |
| string | String of length zero | string |
| Hashtable | Empty object | None |
| Any array type | Each element takes on 0 cast to that type | A comma-separated set of integers that specify each dimension size, in row-major order. |

Windows PowerShell: Some value types (such as System.Decimal) may have constructors, in which case, arguments can be passed to those constructors via ArgumentList.

Windows PowerShell: The set of values specified with ArgumentList is the argument list that matches a constructor for the type specified. TypeName can be any non-abstract .NET or user-defined type.

Parameters:

-ArgumentList <object[]> — Specifies a list of arguments that are used in the object's creation. Args is an alias for ArgumentList.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Position 2 | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-ComObject <string> — Specifies the programmatic identifier (ProgID) of the COM object.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Property <hashtable> — Sets property values and invokes methods of the new object. In the Hashtable, the keys are the names of properties or methods and the values are property values or method arguments, respectively. New-Object creates the object, sets each property value, and invokes each method, in the order in which they appear in the hash table.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

Windows PowerShell: If the new object is derived from type PsObject, and a property is specified that does not exist on that object, New-Object adds the specified property to the object as a NoteProperty. If the new object type is not derived from PsObject, an unspecified non-terminating error is produced.

-Strict [ <SwitchParameter> ] — Specifies that an error should be raised if the COM object uses an interop assembly. This enables actual COM objects to be distinguished from .NET Framework objects with COM-callable wrappers.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

Windows PowerShell: Parameter Strict.

-TypeName <string> — Specifies the name of the type.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

None.

Outputs:

The cmdlet returns a reference to the new object. If it fails, New-Object throws an exception of unspecified type.

Examples:

New-Object 'bool'  
New-Object 'string' 'A red house'  
New-Object 'int[]' 0  
New-Object 'double[,]' 3,2  
New-Object -ArgumentList 2,4,3 -TypeName 'int[,,]'

## New-Variable (alias nv)

Synopsis:

Creates a new variable.

Syntax:

New-Variable [ -Name ] <string> [ [ -Value ] <object> ] [ -Description <string> ]  
[ -Force ] [ -Option { None | ReadOnly | Constant | Private | AllScope } ]  
[ -PassThru ] [ -Scope <string> ] [ -Confirm ] [ -WhatIf ] [ <CommonParameters> ]

Description:

This cmdlet creates a new variable, and allows various characteristics of that variable to be specified. (The only characteristics that be specified when a variable is defined in the PowerShell language are initial value and scope.)

Parameters:

-Confirm [ <SwitchParameter> ] — (alias cf) Prompts for confirmation before executing the cmdlet. If confirmation is denied, the cmdlet terminates with the new variable being created or changed.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Description <string> — Sets the value of the Description property (§4.5.3) of the variable.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Force [ <SwitchParameter> ] — This switch parameter allows the value of an existing read-only variable to be changed. For more information, see the Option parameter.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Name <string> — Specifies the name of the new variable. (Unlike the PowerShell language, the name here need not have a leading $.)

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-Option <ScopedItemOptions> — Sets the value of the Options property (§4.5.1) of the new variable. The valid values are:

* None: Sets no options.
* ReadOnly: The value of the variable cannot be changed except by using the Force parameter.
* Constant: The variable cannot be deleted, and its characteristics cannot be changed.
* Private: The variable is available only within the scope in which it is defined, and not in any child scopes. (This is equivalent to using the PowerShell language scope modifier private.)
* AllScope: The variable is copied to any new scopes that are created, but is not put into any existing scopes.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: "None" |
| Accept pipeline input: No | | Accept wildcard characters: No |

-PassThru [ <SwitchParameter> ] — Causes the cmdlet to write to the pipeline an object that represents the new variable (§4.5.3). (Ordinarily, the cmdlet does not write any output to the pipeline.)

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Scope <string> — Specifies the scope (3.5) of the new variable. The valid values are "Global", "Script", and "Local", or scope number (§3.5.2).

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: "Local" |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Value <object> — Specifies the initial value of the variable. If no initial value is specified, the variable is created without an initial value. (From within PowerShell, the variable will test True against $null.)

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Position 2 | Default value: None |
| Accept pipeline input: Yes, ByValue or ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-WhatIf [ <SwitchParameter> ] — (alias wi) Describes what would happen if the cmdlet were executed without actually executing it.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

Any kind of value can be written to the pipeline to New-Variable.

Outputs:

None unless the -PassThru switch parameter is used.

Examples:

# The following are equivalent; they all create a variable called $Count1  
  
New-Variable -Name 'Count1' -Value 100  
New-Variable -Name 'Count1' 100  
New-Variable 'Count1' -Value 100  
New-Variable 'Count1' 100

$name = 'Count1'; $value = 100  
New-Variable $name $value  
$Count1 = 100

New-Variable -Name Count30 -Value 150 -Option ReadOnly  
$Count30 = -40 # rejected as $Count30 is read-only  
  
New-Variable -Name Count30 -Value 151 -Force # overwrites Count30

New-Variable -Name Count51 -Value 200 -PassThru | CommandX

New-Variable -Name Count61 -Value 200 -Scope "Script"  
New-Variable -Name Count64 -Value 200 -Scope "0" # local scope

New-Variable -Name Count70 -Value 150 -Option Constant  
$Count70 = -40 # rejected as $count70 is not writable

New-Variable -Name Count71 -Value 150 -Option Private  
New-Variable -Name Count72 -Value 150 -Option AllScope  
New-Variable -Name Count80 -Value 150 -Confirm  
New-Variable -Name Count90 -Value 150 -WhatIf

## Pop-Location (alias popd)

Synopsis:

Sets the current working location (§4.5.5) to that on the top of the specified working location stack.

Syntax:

Pop-Location [ -PassThru ] [ -StackName *<string>* ] [ <CommonParameters> ]

Description:

This cmdlet sets the current working location to that on the top of the specified working location stack, and removes that location from that stack.

Parameters:

-PassThru [ <SwitchParameter> ] —This parameter causes the cmdlet to write to the pipeline an object that represents the working location (§3.1.4) popped from the stack. (Ordinarily, the cmdlet does not write any output to the pipeline.)

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-StackName <string> — Specifies the location stack from which the working location is to be popped. $null and "" both indicate the current working location stack. "default" indicates the default working location stack at session startup. If this parameter is omitted, the current working location stack is used.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

None.

Outputs:

None unless the PassThru switch parameter is used. If the stack is empty, $null is returned.

Examples:

Pop-Location  
$v = Pop-Location -StackName "Stack3" -PassThru

## Push-Location (alias pushd)

Synopsis:

Adds the current or a specified working location to the top of a given working location stack.

Syntax:

Push-Location [ -LiteralPath ] *<string>* [ -PassThru ] [ -StackName *<string>* ]  
[ <CommonParameters> ]

Push-Location [ [ -Path ] *<string>* ] [ -PassThru ] [ -StackName *<string>* ]  
[ <CommonParameters> ]

Description:

This cmdlet pushes the current or a specified working location onto a location stack. Optionally, it can also change the current working location. If the stack does not exist, it is created.

Parameters:

-LiteralPath <string> — (alias PSPath) Specifies a path to be pushed to the stack and to be set as the current working location. The string is used exactly as it is written. Characters that look like wildcards are not interpreted as such.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-PassThru [ <SwitchParameter> ] —This parameter causes the cmdlet to write to the pipeline an object that represents the working location (§3.1.4) pushed onto the stack. (Ordinarily, the cmdlet does not write any output to the pipeline.)

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Path <string> — Specifies a path to be pushed to the stack and to be set as the current working location.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByValue and ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-StackName <string> — Specifies the location stack to which the working location is to be pushed. $null and "" both indicate the current working location stack. "default" indicates the default working location stack at session startup. If this parameter is omitted, the current working location stack is used.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

A string containing a path (but not a literal path) can be written to the pipeline to Push-Location.

Outputs:

None unless the -PassThru switch parameter is used.

Examples:

$v = Push-Location -PassThru  
Push-Location -Path "E:\temp" -StackName "Stack3"

## Remove-Item (alias del, erase, rd, ri, rm, rmdir)

Synopsis:

Deletes one or more items (§3.3).

Syntax:

Remove-Item [ -LiteralPath ] <string[]> [ -Credential <Credential> ]  
[ -Exclude <string[]> ] [ -Filter <string> ] [ -Force ] [ -Include <string[]> ]  
[ -Recurse ] [ -Confirm ] [ -WhatIf ] [ <CommonParameters> ]

Remove-Item [ -Path ] <string[]> [ -Credential <Credential> ]  
[ -Exclude <string[]> ] [ -Filter <string> ] [ -Force ] [ -Include <string[]> ]  
[ -Recurse ] [ -Confirm ] [ -WhatIf ] [ <CommonParameters> ]

Description:

This cmdlet deletes one or more items.

Parameters:

-Confirm [ <SwitchParameter> ] — (alias cf) Prompts for confirmation before executing the cmdlet. If confirmation is denied, the cmdlet terminates with the new variable being created or changed.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Credential <Credential> — Specifies a user account that has permission to perform this action. The default is the current user. See §4.5.23

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-ExcludePath <string[]> — Specifies the path(s) to exclude from the operation.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Filter <string> — Specifies a file filter in the provider's format or language. This parameter qualifies the Path parameter. The syntax of the filter, including the use of wildcards, depends on the provider.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Force [ <SwitchParameter> ] — Allows the cmdlet to remove items that cannot otherwise be changed, such as hidden or read-only files or read-only aliases or variables. The cmdlet cannot remove constant aliases or variables. Implementation varies from provider to provider.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-IncludePath <string[]> — Specifies the path(s) to include in the operation.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-LiteralPath <string[]> — (alias PSPath) Specifies the path(s) to the item(s) to be deleted. The strings are used exactly as they are written. Characters that look like wildcards are not interpreted as such.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Path <string[]> — Specifies the path(s) to the item(s) to be deleted.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByValue or ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Recurse [ <SwitchParameter> ] — Specifies a recursive delete.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-WhatIf [ <SwitchParameter> ] — (alias wi) Describes what would happen if the cmdlet were executed without actually executing it.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

A path (but not literal path) can be written to the pipeline to Remove-Item.

Outputs:

None

Examples:

Remove-Item C:\Test\\*.\* -Recurse  
Remove-Item \* -Include \*.doc -Exclude \*1\*  
Remove-Item -Path C:\Test\hidden-RO-file.txt -force

## Remove-Module (alias rmo)

Synopsis:

Removes one or more modules (§3.14) from the current session.

Syntax:

Remove-Module [ -ModuleInfo ] <PSModuleInfo[]> [ -Force ] [ -Confirm ]  
[ -WhatIf ] [ <CommonParameters> ]

Remove-Module [ -Name ] <string[]> [ -Force ] [ -Confirm ] [ -WhatIf ]  
[ <CommonParameters> ]

Description:

This cmdlet removes the members of a module from the current session.

Windows PowerShell: If the module includes an assembly (.dll), all members that are implemented by the assembly are removed, but the assembly is not unloaded.

Parameters:

-Confirm [ <SwitchParameter> ] — Prompts for confirmation before executing the cmdlet. If confirmation is denied, the cmdlet terminates with the new variable being created or changed.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Force [ <SwitchParameter> ] — Removes modules even when their access mode is read-only.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-ModuleInfo < PSModuleInfo[]> — Specifies the module objects (§4.5.12) to remove.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByValue (§12.3.7) | | Accept wildcard characters: No |

-Name <string[]> — Specifies the names of the modules to remove.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByValue (§12.3.7) | | Accept wildcard characters: Yes |

-WhatIf [ <SwitchParameter> ] — (alias wi) Describes what would happen if the cmdlet were executed without actually executing it.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

A name string or module objects can be written to the pipeline to Remove-Module.

Outputs:

None

Examples:

Remove-Module PSTest\_Temperature  
Remove-Module Lib1,Lib2

## Remove-Variable (alias rv)

Synopsis:

Deletes one or more variables.

Syntax:

Remove-Variable [ -Name ] <string[]> [ -Exclude *<string[]>* ] [ -Force ]  
[ -Include *<string[]>* ] [ -Scope <string> ]   
[ -Confirm ] [ -WhatIf ] [ <CommonParameters> ]

Description:

This cmdlet deletes one or more variables from the specified scope. This cmdlet cannot delete variables that are set as constants or those that are owned by the PowerShell runtime.

Parameters:

-Confirm [ <SwitchParameter> ] — (alias cf) Prompts for confirmation before executing the cmdlet. If confirmation is denied, the cmdlet terminates with the new variable being created or changed.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Exclude <string[]> — Specifies the name(s) to be excluded from the operation.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: Yes |

-Force [ <SwitchParameter> ] — This switch parameter allows an existing read-only variable to be deleted.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Include <string[]> — Specifies the name(s) to be included in the operation.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: Yes |

-Name <string[]> — Specifies the name(s) of the variables to be deleted.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Scope <string> — Specifies the scope (§3.5) of the variable to be deleted. The valid values are "Global", "Script", and "Local", or scope number (§3.5.2).

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: "Local" |
| Accept pipeline input: No | | Accept wildcard characters: No |

-WhatIf [ <SwitchParameter> ] — (alias wi) Describes what would happen if the cmdlet were executed without actually executing it.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

An object that represents a variable (§4.5.3) can be written to the pipeline to Remove-Variable.

Outputs:

None.

Examples:

Remove-Variable 'Count10?' -Exclude 'Count101','Count102'

## Rename-Item (alias ren, rni)

Synopsis:

Renames an item (§3.3).

Syntax:

Rename-Item [ -Path ] <string> [ -NewName ] <string>   
[ -Credential <Credential> ] [ -PassThru ] [ -Force ] [ -Confirm ] [ -WhatIf ]  
[ <CommonParameters> ]

Description:

This cmdlet changes the name of a specified item. (Rename-Item cannot be used to move an item.)

Parameters:

-Confirm [ <SwitchParameter> ] — (alias cf) Prompts for confirmation before executing the cmdlet. If confirmation is denied, the cmdlet terminates with the new variable being created or changed.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Credential <Credential> — Specifies a user account that has permission to perform this action. The default is the current user. See §4.5.23

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-Force [ <SwitchParameter> ] — Allows the cmdlet to rename items that cannot otherwise be changed, such as hidden or read-only files or read-only aliases or variables. The cmdlet cannot change constant aliases or variables. Implementation varies from one provider to another.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-NewName <string> — Specifies the new name of the item. Enter only a name, not a path and name.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 2 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-PassThru [ <SwitchParameter> ] — Returns an object that represents the renamed item. By default, this cmdlet does not generate any output.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Path <string> — (alias PSPath) Specifies the path to the item to be renamed.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByValue or ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-WhatIf [ <SwitchParameter> ] — (alias wi) Describes what would happen if the cmdlet were executed without actually executing it.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

A path can be written to the pipeline to Rename-Item.

Outputs:

If PassThru is present, an object (§4.5) that describes the item renamed.

Examples:

Rename-Item J:\Test\File1.txt File1.tmp

## Resolve-Path (alias rvpa)

Resolves the wildcard characters in one or more paths (§3.4), and outputs the path(s) contents.

Syntax:

Resolve-Path [ -LiteralPath ] <string[]> [ -Credential <Credential> ]  
[ -Relative ] [ <CommonParameters> ]

Resolve-Path [ -Path ] <string[]> [ -Credential <Credential> ]  
[ -Relative ] [ <CommonParameters> ]

Description:

This cmdlet interprets the wildcard characters in one or more paths and outputs the items and containers at the location specified by the path(s).

Parameters:

-Credential <Credential> — Specifies a user account that has permission to perform this action. The default is the current user. See §4.5.23

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-LiteralPath <string[]> — (alias PSPath) Specifies the paths to be resolved. The strings are used exactly as they are written. Characters that look like wildcards are not interpreted as such.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-Path <string[]> — Specifies the paths to be resolved.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByName or ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Relative [ <SwitchParameter> ] — Retrieves a relative path.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

A string containing a path (but not a literal path) can be written to the pipeline to Test-Path.

Outputs:

Relative is specified: If the result is only one path, a string is output. If multiple paths result, an unconstrained 1-dimensional array of string is output, where the value of each element corresponds to the path provided.

Relative is not specified: If the result is only one path, an object that describes a path (§4.5.5) is output. If multiple paths result, an unconstrained 1-dimensional array of objects that describe paths is output, where the value of each element corresponds to the path provided.

Examples:

Resolve-Path -Path E:\Temp\W\*,E:\Temp\Action\*  
Resolve-Path -Path E:\Temp\Action\* -Relative

## Select-Object (alias select)

Synopsis:

Selects specified properties of an object or set of objects. It can also select unique objects from an array of objects, or it can select a specified number of objects from the beginning or end of an array of objects.

Syntax:

Select-Object [ [ -Property ] <object[]> ] [ -ExcludeProperty <string[]> ]  
[ -ExpandProperty <string> ] [ -First <int> ] [ -InputObject <object> ]  
[ -Last <int> ] [ -Skip <int> ] [ -Unique ] [ CommonParameters> ]

Select-Object [ -Index <int[]> ] [ -InputObject <object> ] [ -Unique ]  
[ CommonParameters> ]

Description:

This cmdlet gets the specified properties of an object or set of objects. It can also select unique objects from an array of objects, or it can select a specified number of objects from the beginning or end of an array of objects.

When specific properties are selected, the values of those properties are copied from the input objects to new objects that have the specified properties and copied values. The Property parameter specifies the properties to select. Alternatively, the First, Last, Unique, Skip, and Index parameters are used to select particular objects from an array of input objects.

Parameters:

-ExcludeProperty [ <string[]> ] — Removes the specified properties from the selection. This parameter is effective only when the command also includes the Property parameter.

Ordinarily, a property is designated by a string containing the property's name. However, a property can be a value that is computed rather than being an actual named property in the object. To accommodate this, the value of the Expression key can be a script block that computes the key value, and the value of the Name or Label key is a string naming the computed property, as in @{Name="Start Day"; Expression={…}}.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: Yes |

-ExpandProperty [ <string> ] — Specifies a property to select, and indicates that an attempt should be made to expand that property. For example, if the specified property is an array, each value of the array is included in the output. If the property contains an object, the properties of that object are displayed in the output.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: Yes |

-First [ <int> ] — Specifies the number of objects to select from the beginning of an array of input objects.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Index [ <int[]> ] — Selects objects from an array based on their index values.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-InputObject <object> — Specifies objects to write to the cmdlet through the pipeline.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByValue (§12.3.7) | | Accept wildcard characters: No |

-Last [ <int> ] — Specifies the number of objects to select from the end of an array of input objects.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Property [ <object[]> ] — Specifies the properties to select. Wildcards are permitted.

The value of the Property parameter can be a new calculated property (see the ExcludeProperty parameter for details.).

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: Yes |

-Skip [ <int> ] — Does not select the specified number of items. By default, this parameter counts from the beginning of the array or list of objects, but if the command uses the Last parameter, it counts from the end of the list or array. Unlike the Index parameter, which starts counting from 0, the Skip parameter begins at 1.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Unique [ <SwitchParameter> ] — Specifies that if a subset of the input objects has identical properties and values, only a single member of the subset will be selected.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

Any object type can be piped to this cmdlet.

Outputs:

A single object or an unconstrained 1-dimensional array of objects, as selected. If specific properties are chosen, the output will be a custom object (§4.5.13).

Examples:

"b", "a","b","a","a","c","a" | Select-Object -Unique  
Get-Date | Select-Object -Property Year,Month,Day

$dates = $d1,$d2,$d3,$d4,$d5 # a collection of 5 date/time values  
$dates | Select-Object -Property Year,Month,Day  
$dates | Select-Object -First 1 -Last 2

## Set-Alias (alias sal)

Sets one or more characteristics of one or more existing aliases. If the aliases do not exist, creates them with those characteristics.

Syntax:

Set-Alias [ -Name ] <string> [ [ -Value ] <string> ] [ -Description <string> ]  
[ -Force ] [ -Option { None | ReadOnly | Constant | Private | AllScope } ]  
[ -PassThru ] [ -Scope <string> ] [ -Confirm ] [ -WhatIf ] [ <CommonParameters> ]

Description:

This cmdlet sets one or more characteristics of one or more existing aliases. If the aliases do not exist, creates them with those characteristics.

Parameters:

-Confirm [ <SwitchParameter> ] — (alias cf) Prompts for confirmation before executing the cmdlet. If confirmation is denied, the cmdlet terminates with the new variable being created or changed.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Description <string> — Specifies a description of the alias.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Force [ <SwitchParameter> ] — This switch parameter allows an existing alias to be reassigned and/or its characteristics to be changed. See -Option ReadOnly.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Name <string> — Specifies the name of the new alias or the alias whose characteristics are to be changed.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-Option <ScopedItemOptions> — Sets the value of the Options property of the new alias. The valid values are:

* None: Sets no options.
* ReadOnly: The alias cannot be reassigned and its characteristics cannot be changed except by using the Force parameter.
* Constant: The alias cannot be deleted, and its characteristics cannot be changed.
* Private: The alias is available only within the scope in which it is defined, and not in any child scopes.
* AllScope: The alias is copied to any new scopes that are created, but is not put into any existing scopes.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: "None" |
| Accept pipeline input: No | | Accept wildcard characters: No |

-PassThru [ <SwitchParameter> ] — Causes the cmdlet to write to the pipeline an object that represents the new alias (§4.5.4). (Ordinarily, the cmdlet does not write any output to the pipeline.)

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Scope <string> — Specifies the scope (§3.5) of the new alias. The valid values are "Global", "Script", and "Local", or scope number (§3.5.2).

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: "Local" |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Value <string> — Specifies the name of the cmdlet or command element that is being aliased.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 2 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-WhatIf [ <SwitchParameter> ] — (alias wi) Describes what would happen if the cmdlet were executed without actually executing it.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

None.

Outputs:

None unless the PassThru switch parameter is used.

Examples:

New-Alias 'Func1' 'F1'  
Set-Alias 'Func1' 'F1' -Description "…"  
Set-Alias 'Func1' 'F1' -Scope Global  
  
function F1 { … }

## Set-Content (alias sc)

Synopsis:

Replaces the content in an item (§3.3) with new content.

Syntax:

Set-Content [ -LiteralPath ] <string[]> [ -Value <object[]> ]  
[ -Credential <Credential> ] [ -Exclude <string[]> ] [ -Filter <string> ]  
[ -Force ] [ -Include <string[]> ] [ -PassThru ] [ -Confirm ] [ -WhatIf ]  
[ <CommonParameters> ]

Set-Content [ -Path ] <string[]> [ -Value <object[]> ] [ -Credential <Credential> ]  
[ -Exclude <string[]> ] [ -Filter <string> ] [ -Force ] [ -Include <string[]> ]  
[ -PassThru ] [ -Confirm ] [ -WhatIf ] [ <CommonParameters> ]

Description:

This cmdlet replaces the content in the specified item.

Parameters:

-Confirm [ <SwitchParameter> ] — (alias cf) Prompts for confirmation before executing the cmdlet. If confirmation is denied, the cmdlet terminates with the new variable being created or changed.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Credential <Credential> — Specifies a user account that has permission to perform this action. The default is the current user. See §4.5.23

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-Exclude <string[]> — Specifies the path(s) to be excluded from the operation.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Filter <string> — Specifies a file filter in the provider's format or language. This parameter qualifies the Path parameter. The syntax of the filter, including the use of wildcards, depends on the provider.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Force [ <SwitchParameter> ] — This switch parameter allows the cmdlet to write over an existing item. However, this parameter cannot be used to override security restrictions.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Include <string[]> — Specifies the path(s) to be included in the operation.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-LiteralPath <string[]> — (alias PSPath) Specifies the path(s) of the item. The strings are used exactly as they are written. Characters that look like wildcards are not interpreted as such.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-PassThru [ <SwitchParameter> ] — Returns an object or an array of objects representing the replaced content. By default, this cmdlet does not generate any output.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Path <string[]> — Specifies the path(s) of the item.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Value <object[]> — Specifies the replacement values.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByValue or ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-WhatIf [ <SwitchParameter> ] — (alias wi) Describes what would happen if the cmdlet were executed without actually executing it.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

Any kind of objects for the Value parameter can be written to the pipeline to Add-Content. However, the object is converted to a string before it is added to the item.

Outputs:

None unless the PassThru switch parameter is used.

Examples:

Set-Content "J:\Test\File2.txt" -Value "Line A","Line B"

## Set-Item (alias si)

Synopsis:

Changes the value of one or more items (§3.3).

Syntax:

Set-Item [ -LiteralPath ] <string[]> [ [ -Value ] <object> ]  
[ -Credential <Credential> ] [ -Exclude <string[]> ] [ -Filter <string> ]  
[ -Force ] [ -Include <string[]> ] [ -PassThru ] [ -Confirm ] [ -WhatIf ]  
[ <CommonParameters> ]

Set-Item [ -Path ] <string[]> [ [ -Value ] <object> ] [ -Credential <Credential> ]  
[ -Exclude <string[]> ] [ -Filter <string> ] [ -Force ] [ -Include <string[]> ]  
[ -PassThru ] [ -Confirm ] [ -WhatIf ] [ <CommonParameters> ]

Description:

This cmdlet changes the value of one or more items. Regarding changing the value of an alias, see Set-Alias (§13.46).Regarding changing the value of a Variable, see Set-Variable (§13.37).

Parameters:

-Confirm [ <SwitchParameter> ] — (alias cf) Prompts for confirmation before executing the cmdlet. If confirmation is denied, the cmdlet terminates with the new variable being created or changed.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Credential <Credential> — Specifies a user account that has permission to perform this action. The default is the current user. See §4.5.23

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-Exclude <string[]> — Specifies the paths to exclude from the operation.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Filter <string> — Specifies a file filter in the provider's format or language. This parameter qualifies the Path parameter. The syntax of the filter depends on the provider.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Force [ <SwitchParameter> ] — This switch parameter allows the cmdlet to set items that cannot otherwise be changed, such as read-only alias or variables. The cmdlet cannot change constant aliases or variables, however.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Include <string[]> — Specifies the paths to include in the operation.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-LiteralPath <string[]> — (alias PSPath) Specifies the paths of one or more items. The strings are used exactly as they are written. Characters that look like wildcards are not interpreted as such.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-PassThru [ <SwitchParameter> ] — Passes one or more objects representing the items to the pipeline. By default, this cmdlet does not generate any output.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Path <string[]> — Specifies the paths of one or more items.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Value <object> — Specifies the value to be used to initialize the item(s). For a function, the value is a script block that implements the function.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Position 2 | Default value: None |
| Accept pipeline input: Yes, ByValue or ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-WhatIf [ <SwitchParameter> ] — (alias wi) Describes what would happen if the cmdlet were executed without actually executing it.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

Any kind of object for the Value parameter can be written to the pipeline to Set-Item.

Outputs:

If PassThru is specified, a single object (§4.5) or an unconstrained 1-dimensional array of objects that describes the item(s) whose value was set.

Examples:

Set-Item Variable:Count -Value 200  
Set-Item Env:Day1,Env:Day2 -Value "Monday" -PassThru

## Set-Location (alias cd, chdir, sl)

Synopsis:

Sets the current working location or sets the current working location stack.

Syntax:

Set-Location [ -LiteralPath ] *<string>* [ -PassThru ] [ <CommonParameters> ]

Set-Location [ [ -Path ] *<string>* ] [ -PassThru ] [ <CommonParameters> ]

Set-Location [ -StackName *<string>* ] [ -PassThru ] [ <CommonParameters> ]

Description:

The first two forms of invoking this cmdlet set the current working location to the specified location. The third form sets the current working location stack to the specified stack.

Parameters:

-LiteralPath <string> — (alias PSPath) Specifies a path to the new working location. The string is used exactly as it is written. Characters that look like wildcards are not interpreted as such.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-PassThru [ <SwitchParameter> ] — For the first two forms of invoking this cmdlet this parameter causes the cmdlet to write to the pipeline an object that represents a working location (§3.1.4). For the third form, this parameter causes the cmdlet to write to the pipeline a single stack of working location objects (§3.1.4). (Ordinarily, the cmdlet does not write any output to the pipeline.)

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Path <string> — Specifies a path to the new working location.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByValue and ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-StackName <string> — Specifies the location stack to be set to the current working location stack. $null and "" both indicate the current working location stack. "default" indicates the default working location stack at session startup.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

A string containing a path (but not a literal path) can be written to the pipeline to Set-Location.

Outputs:

None unless the PassThru switch parameter is used.

Examples:

Set-Location c:\temp  
Set-Location G:  
Set-Location -StackName "Stack1" -PassThru

## Set-Variable (alias set, sv)

Synopsis:

Sets one or more characteristics of one or more existing variables or, if the variables do not exist, creates them with those characteristics.

Syntax:

Set-Variable [ -Name ] <string[]> [ [ -Value ] <object> ]  
[ -Description <string> ] [ -Exclude *<string[]>* ] [ -Force ] [ -Include *<string[]>* ]  
[ -Option { None | ReadOnly | Constant | Private | AllScope } ]  
[ -PassThru ] [ -Scope <string> ] [ -Confirm ] [ -WhatIf ] [ <CommonParameters> ]

Description:

This cmdlet sets one or more characteristics of one or more existing variables or, if the variables do not exist, creates them with those characteristics. (The only characteristics that be specified when a variable is defined in the PowerShell language are initial value and scope.)

Parameters:

-Confirm [ <SwitchParameter> ] — (alias cf) Prompts for confirmation before executing the cmdlet. If confirmation is denied, the cmdlet terminates with the new variable being created or changed.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Description <string> — Sets the value of the Description property (§4.5.3) of the variable.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Exclude <string[]> — Specifies the name(s) to be excluded from the operation.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: Yes |

-Force [ <SwitchParameter> ] — This switch parameter allows the value of an existing read-only variable to be changed. For more information, see the Option parameter.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Include <string[]> — Specifies the name(s) to be included in the operation.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: Yes |

-Name < string[]> — Specifies the name(s) of the existing or new variable(s). (Unlike the PowerShell language, the name(s) here need not have a leading $.)

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Option <ScopedItemOptions> — Sets the value of the Options property (§4.5.3) of the variable. The valid values are:

* None: Sets no options.
* ReadOnly: The value of the variable cannot be changed except by using the Force parameter. (A ReadOnly variable can be deleted via Remove-Variable.)
* Constant: The variable cannot be deleted, and its characteristics cannot be changed. This option can only be used if the variable is created by this cmdlet. If a variable by that name already exists in the given scope, it cannot be made Constant by this cmdlet.
* Private: The variable is available only within the scope in which it is defined, and not in any child scopes. (This is equivalent to using the PowerShell language scope modifier private.)
* AllScope: The variable is copied to any new scopes that are created, but is not put into any existing scopes.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: "None" |
| Accept pipeline input: No | | Accept wildcard characters: No |

-PassThru [ <SwitchParameter> ] — Causes the cmdlet to write to the pipeline an object that represents the new variable (§4.5.3). (Ordinarily, the cmdlet does not write any output to the pipeline.)

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Scope <string> — Specifies the scope (§3.5) of the new variable. The valid values are "Global", "Script", "Local", or scope number (§3.5.2).

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: "Local" |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Value <object> — Specifies the value of the variable(s). If no value is specified, the value of an existing variable is unchanged, and a newly created variable has no initial value.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Position 2 | Default value: None |
| Accept pipeline input: Yes, ByValue or ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-WhatIf [ <SwitchParameter> ] — (alias wi) Describes what would happen if the cmdlet were executed without actually executing it.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

An object that represents a variable can be written to the pipeline to Set-Variable.

Outputs:

None unless the PassThru switch parameter is used.

Examples:

Set-Variable 'Count100' 100 # create new  
Set-Variable 'Count101','Count102','Count103' 110 # create new

Set-Variable 'Count100' 200 # change existing  
Set-Variable 'Count101','Count102','Count104' 210 # change 2, create 1

Set-Variable 'Count10?' 111 -Exclude 'Count101','Count102'

## Sort-Object (alias sort)

Synopsis:

Sorts objects by one or more property values.

Syntax:

Sort-Object [ [ -Property ] <object[]> ] [ -CaseSensitive ] [ -Unique ]  
[ -Culture <string> ] [ -Descending ] [ -InputObject <object> ]  
[ CommonParameters> ]

Description:

This cmdlet sorts objects in ascending or descending order based on the values of one or more properties of the object.

If an object does not have a specified property, the property value for that object is assumed to be NULL and it is placed at the end of the sort order.

The property values are compared using the Compare method for that type, if one exists. Otherwise, the property value is converted to a string and string comparison is used.

Parameters:

-CaseSensitive [ <SwitchParameter> ] — Indicates that the sort should be case sensitive. By default, sorting is not case sensitive.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Culture <string> — Specifies the cultural configuration to use when sorting (as in "en-US" for US English using language codes from ISO 639-1 and country codes from ISO 3166-1).

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Descending [ <SwitchParameter> ] — Sorts the objects in descending order. The default is ascending order. This parameter applies to all properties. To sort some properties in ascending order and others in descending order, see the Property parameter.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-InputObject <object> — Specifies the objects to be sorted. When this parameter is used, Sort-Object receives one object that represents the collection. Because one object cannot be sorted, Sort-Object returns the entire collection unchanged. To sort objects, pipe them to Sort-Object.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByValue (§12.3.7) | | Accept wildcard characters: No |

-Property <object[]> — Specifies the properties to use when sorting. If multiple properties are specified, the objects are sorted by the first property. If more than one object has the same value for the first property, those objects are sorted by the second property. This process continues for the remaining properties, if any. If no properties are specified, the cmdlet sorts based on the default properties for the object type.

To sort different properties in different orders express each object in the array as a hash literal. Each hash literal must contain a key called Expression whose value designates the property, and a key called Ascending or Descending whose bool value to specify the sort order.

Ordinarily, a property is designated by a string containing the property's name. However, a property can be a value that is computed rather than being an actual named property in the object. To accommodate this, the value of the Expression key can be a script block that computes the key value.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: Yes |

-Unique [ <SwitchParameter> ] — Eliminates duplicates and returns only the unique members of the collection.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

The objects to be sorted.

Outputs:

The objects in the desired sorted order.

Examples:

$d1 = Get-Date "03/04/2006 6:15:23 AM"   
…  
$d5 = Get-Date "03/01/2006 11:23:56 PM"   
$dates = $d1,$d2,$d3,$d4,$d5  
  
$v = $dates | Sort-Object  
$v = $dates | Sort-Object -Descending -Unique  
$v = $dates | Sort-Object -Property @{Expression="Day" ;  
 Ascending=$true},@{Expression="Hour" ; Ascending=$false}

Windows PowerShell: If a sort key property is an enumeration, the enumeration values are sorted in numeric order.

## Split-Path

Synopsis:

Retrieves the specified part of one or more paths (§3.4) or reports on certain characteristics of those paths.

Syntax:

Split-Path [ -IsAbsolute ] [ -Path ] <string[]> [ -Credential <Credential> ]  
[ -LiteralPath <string[]> ] [ -Resolve ] [ <CommonParameters> ]

Split-Path [ -Leaf ] [ -Path ] <string[]> [ -Credential <Credential> ]  
[ -LiteralPath <string[]> ] [ -Resolve ] [ <CommonParameters> ]

Split-Path [ -NoQualifier ] [ -Path ] <string[]> [ -Credential <Credential> ]  
[ -LiteralPath <string[]> ] [ -Resolve ] [ <CommonParameters> ]

Split-Path [ -Parent ] [ -Path ] <string[]> [ -Credential <Credential> ]  
[ -LiteralPath <string[]> ] [ -Resolve ] [ <CommonParameters> ]

Split-Path [ -Qualifier ] [ -Path ] <string[]> [ -Credential <Credential> ]  
[ -LiteralPath <string[]> ] [ -Resolve ] [ <CommonParameters> ]

Description:

This cmdlet retrieves the specified part of a path, such as the parent directory, a child directory, or a file name. It also can retrieve the items that are referenced by the split path, and indicate whether the path is relative or absolute.

Parameters:

-Credential <Credential> — Specifies a user account that has permission to perform this action. The default is the current user. See §4.5.23

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-IsAbsolute [ <SwitchParameter> ] — Determines if the path is absolute.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Leaf [ <SwitchParameter> ] — Retrieves the last item or container in the path.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-LiteralPath <string[]> — Specifies one or more paths to be split. The strings are used exactly as they are written. Characters that look like wildcards are not interpreted as such.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-NoQualifier [ <SwitchParameter> ] — Returns the path without the qualifier.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-Parent [ <SwitchParameter> ] — Retrieves the parent containers of the item or container specified by the path. This parameter is the default split location parameter.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-Path <string[]> — Specifies one or more paths to be split.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByName or ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-Qualifier [ <SwitchParameter> ] — Retrieves the qualifier of the specified path.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Position 2 | Default value: $true |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-Resolve [ <SwitchParameter> ] — Retrieves the items referenced by the resulting split path instead of retrieving the path elements themselves.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

A string containing a path (but not a literal path) can be written to the pipeline to Split-Path.

Outputs:

For IsAbsolute: If only one path is provided, $true is output if the path is absolute, and $false otherwise. If multiple paths are provided, an unconstrained 1-dimensional array of bool values is output. The value of each element is $true if the corresponding path is absolute, and $false otherwise.

For Leaf, NoQualifier, Parent, and Qualifier: If only one path is provided, a string is output. If multiple paths are provided, an unconstrained 1-dimensional array of string values is output, whose element correspond to the paths provided.

Examples:

Split-Path -Path ..\Temp -IsAbsolute # False  
Split-Path -Path J:\,J:\main\sub1\sub2 -Leaf # "J:\","sub2"  
Split-Path -Path J:\,J:\main\sub1\sub2 -Parent # "",J:\main\sub1"  
Split-Path -Path J:\ -Qualifier # "J:"  
Split-Path -Path J:\,J:\main\sub1\sub2 -NoQualifier # "\","\main\sub1\sub2"

## Tee-Object (alias tee)

Synopsis:

Saves command output in a file or variable as well as writing it to the pipeline.

Syntax:

Tee-Object [ -FilePath ] <string> [ -InputObject <object> ]  
[ CommonParameters> ]

Tee-Object -Variable <string> [ -InputObject <object> ] [ CommonParameters> ]

Description:

This cmdlet writes the output of a command in two directions (like the letter "T"). It stores the output in a file or variable, and writes it to the pipeline.

Parameters:

-FilePath <string> — (alias PSPath) Specifies the file where the cmdlet stores the object.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: Yes, provided it resolves to a single file |

-InputObject <object> — Specifies the object to be replicated.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByValue (§12.3.7) | | Accept wildcard characters: No |

-Variable <string> — Specifies the variable (without a leading $) where the cmdlet stores the object.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

The object to be copied.

Outputs:

The input object.

Examples:

123 | Tee-Object -FilePath "E:\Temp\Log.txt"  
"red" | Tee-Object -Variable Copy

## Test-Path

Synopsis:

Determines whether all elements of a path (§3.4) exist.

Syntax:

Test-Path [ -LiteralPath ] <string[]> [ -Credential <Credential> ]  
[ -Exclude <string[]> ] [ -Filter <string> ] [ -Include <string[]> ]  
[ -IsValid ] [ -PathType *<TestPathType>* ] [ <CommonParameters> ]

Test-Path [ -Path ] <string[]> [ -Credential <Credential> ] [ -Exclude <string[]> ]  
[ -Filter <string> ] [ -Include <string[]> ] [ -IsValid ]  
[ -PathType *<TestPathType>* ] [ <CommonParameters> ]

Description:

This cmdlet determines whether all elements of the path exist. It can also tell whether the path syntax is valid and whether the path leads to a container or a terminal element.

Parameters:

-Credential <Credential> — Specifies a user account that has permission to perform this action. The default is the current user. See §4.5.23

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-Exclude <string[]> — Specifies paths that are to be omitted from the operation. This parameter qualifies the Path parameter.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: Yes |

-Filter <string> — Specifies a file filter in the provider's format or language. This parameter qualifies the Path parameter. The syntax of the filter depends on the provider (§3.1).

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: Yes |

-Include <string[]> — Specifies paths that are to be included in the operation. This parameter qualifies the Path parameter.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: Yes |

-IsValid [ <SwitchParameter> ] — Determines whether the syntax of the path is correct, regardless of whether the elements of the path exist.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-LiteralPath <string[]> — Specifies one or more paths to be tested. The strings are used exactly as they are written. Characters that look like wildcards are not interpreted as such.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByPropertyName (§12.3.7) | | Accept wildcard characters: No |

-Path <string[]> — Specifies one or more paths to be tested.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: Yes, ByName or ByPropertyName (§12.3.7) | | Accept wildcard characters: Yes |

-PathType <TestPathType> — Tests whether the final element in the path is a container (Container), a leaf (Leaf), or a container or leaf (Any).

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

A string containing a path (but not a literal path) can be written to the pipeline to Test-Path.

Outputs:

If only one path is being tested, $true is output if that path exists, is valid according to IsValid, or matches PathType, and $false otherwise. If multiple paths are being tested, an unconstrained 1-dimensional array of bool values is output. The value of each element is $true if the corresponding path exists, is valid according to IsValid, or matches PathType, and $false otherwise.

Examples:

Test-Path -Path j:\Files  
Test-Path -Path j:\Files\MyFile1.txt -IsValid  
Test-Path -Path j:\Files\File1.txt,j:\Files\File2.txt,j:\Files\File3.txt  
Test-Path -Path j:\Files\\* -Exclude \*.txt

Test-Path -Path j:\Files -PathType Container  
Test-Path -Path Variable:Count  
Test-Path -Path Function:F1  
Test-Path -Path Env:Path

## Where-Object (alias ?, where)

Synopsis:

Creates a filter to determine which input objects are be passed along a command pipeline.

Syntax:

Where-Object [ -FilterScript ] <scriptblock> [ -InputObject <object> ]  
[ <CommonParameters> ]

Description:

This cmdlet selects objects from the set of objects that are passed to it. It uses a script block as a filter and evaluates that script block for each object. For each object input, if the result of the evaluation is $true that object is returned; otherwise, that object is ignored.

Parameters:

-FilterScript <scriptblock> — Specifies the script block that is used to filter the objects.

|  |  |  |
| --- | --- | --- |
| Required: Yes | Position/Named: Position 1 | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-InputObject <object> — Specifies the objects to be filtered.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: Yes, ByValue (§12.3.7) | | Accept wildcard characters: No |

<CommonParameters> — This cmdlet supports the common parameters Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, Verbose, WarningAction, and WarningVariable.

Inputs:

The objects to be filtered.

Outputs:

The objects accepted by the filter.

Examples:

Get-ChildItem "E:\Files\\*.\*" | Where-Object { $\_.Length -le 1000 }

## Common parameters

The common parameters are a set of cmdlet parameters that can be used with any cmdlet. They are implemented by the PowerShell runtime environment itself, not by the cmdlet developer, and they are automatically available to any cmdlet or function that uses the Parameter attribute (§12.3.7) or CmdletBinding attribute (§12.3.5).

Although the common parameters are accepted by any cmdlet, they might not have any semantics for that cmdlet. For example, if a cmdlet does not generate any verbose output, using the Verbose common parameter has no effect.

Several common parameters override system defaults or preferences that can be set via preference variables (§2.3.2.3). Unlike the preference variables, the common parameters affect only the commands in which they are used.

The common parameters are defined below.

-Debug [ <SwitchParameter> ] — (alias db) Displays programmer-level detail about the operation performed by the command. This parameter works only when the command generates a debugging message. This parameter overrides the value of the $DebugPreference variable for the current command.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-ErrorAction <Action-Preference> — (alias ea) Determines how the cmdlet responds to a non-terminating error from the command. This parameter works only when the command generates an error message. This parameter overrides the value of the $ErrorActionPreference variable for the current command, and it has no effect on terminating errors (such as missing data, parameters that are not valid, or insufficient permissions) that prevent a command from completing successfully.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: "Continue" |
| Accept pipeline input: No | | Accept wildcard characters: No |

-ErrorVariable [+]<VariableName> — (alias ev) Stores errors about the command in the variable specified by VariableName. The leading + causes the errors to be appended to the variable content, instead of replacing it. Specific errors stored in the variable can be accessed by subscripting that variable.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-OutBuffer <int> — (alias ob) Determines the number of objects to accumulate in a buffer before any objects are written to the pipeline. If this parameter is omitted, objects are written as they are generated. When this parameter is present, the next cmdlet in the pipeline is not called until the number of objects generated equals <int> + 1. Thereafter, it writes all objects as they are generated.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-OutVariable [+]<VariableName> — (alias ov) Stores output objects from the command in the variable specified by VariableName. The leading + causes the errors to be appended to the variable content, instead of replacing it. Specific errors stored in the variable can be accessed by subscripting that variable.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

-Verbose [ <SwitchParameter> ] — (alias Args) Displays detailed information about the operation performed by the command. This parameter works only when the command generates a verbose message. This parameter overrides the value of the $VerbosePreference variable for the current command.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: $true |
| Accept pipeline input: No | | Accept wildcard characters: No |

-WarningAction <Action-Preference> — (alias wa) Determines how the cmdlet responds to a warning from the command. This parameter works only when the command generates a warning message. This parameter overrides the value of the $WarningPreference variable for the current command.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: "Continue" |
| Accept pipeline input: No | | Accept wildcard characters: No |

-WarningVariable [+]<VariableName> — (alias wv) Stores warnings about the command in the variable specified by VariableName. The leading + causes the warnings to be appended to the variable content, instead of replacing it. Specific warnings stored in the variable can be accessed by subscripting that variable.

|  |  |  |
| --- | --- | --- |
| Required: No | Position/Named: Named | Default value: None |
| Accept pipeline input: No | | Accept wildcard characters: No |

1. Comment-Based Help

PowerShell provides a mechanism for programmers to document their scripts using special comment directives. Comments using such syntax are called help comments. The cmdlet Get-Help (§13.19) generates documentation from these directives.

* 1. Introduction

A help comment contains a help directive of the form .name followed on one or more subsequent lines by the help content text. The help comment can be made up of a series of single-line-comments or a delimited-comment (§2.2.3). The set of comments comprising the documentation for a single entity is called a help topic.

For example,

# <help-directive-1>  
# <help-content-1>  
…

# <help-directive-n>  
# <help-content-n>

or

<#  
 <help-directive-1>  
 <help-content-1>  
 …

<help-directive-n>  
 <help-content-n>  
#>

All of the lines in a help topic must be contiguous. If a help topic follows a comment that is not part of that topic, there must be at least one blank line between the two.

The directives can appear in any order, and some of the directives may appear multiple times.

Directive names are not case-sensitive.

When documenting a function, help topics may appear in one of three locations:

* Immediately before the function definition with no more than one blank line between the last line of the function help and the line containing the function statement.
* Inside the function's body immediately following the opening curly bracket.
* Inside the function's body immediately preceding the closing curly bracket.

When documenting a script file, help topics may appear in one of two locations:

* At the beginning of the script file, optionally preceded by comments and blank lines only. If the first item in the script after the help is a function definition, there must be at least two blank lines between the end of the script help and that function declaration. Otherwise, the help will be interpreted as applying to the function instead of the script file.
* At the end of the script file.
  1. Help directives
     1. .DESCRIPTION

Syntax:

.DESCRIPTION

Description:

This directive allows for a detailed description of the function or script. (The .SYNOPSIS directive (§A.2.11) is intended for a brief description.) This directive can be used only once in each topic.

Examples:

.DESCRIPTION  
Computes Base to the power Exponent. Supports non-negative integer   
powers only.

* + 1. .EXAMPLE

Syntax:

.EXAMPLE

Description:

This directive allows an example of command usage to be shown.

If this directive occurs multiple times, each associated help content block is displayed as a separate example.

Examples:

.EXAMPLE  
Get-Power 3 4  
81

.EXAMPLE  
Get-Power -Base 3 -Exponent 4  
81

* + 1. .EXTERNALHELP

Syntax:

.EXTERNALHELP <XMLHelpFilePath>

Description:

This directive specifies the path to an XML-based help file for the script or function.

Although comment-based help is easier to implement, XML-based Help is required if more precise control is needed over help content or if help topics are to be translated into multiple languages. The details of XML-based help are not defined by this specification.

Examples:

.ExternalHelp C:\MyScripts\Update-Month-Help.xml

* + 1. .FORWARDHELPCATEGORY

Syntax:

.FORWARDHELPCATEGORY <Category>

Description:

Specifies the help category of the item in ForwardHelpTargetName (§A.2.5). Valid values are Alias, All, Cmdlet, ExternalScript, FAQ, Filter, Function, General, Glossary, HelpFile, Provider, and ScriptCommand. Use this directive to avoid conflicts when there are commands with the same name.

Examples:

See §A.2.5.

* + 1. .FORWARDHELPTARGETNAME

Syntax:

.FORWARDHELPTARGETNAME <Command-Name>

Description:

Redirects to the help topic specified by <Command-Name>.

Examples:

function Help  
{  
<#  
 .FORWARDHELPTARGETNAME Get-Help  
 .FORWARDHELPCATEGORY Cmdlet  
#>  
 …  
}

The command Get-Help help is treated as if it were Get-Help Get-Help instead.

* + 1. .INPUTS

Syntax:

.INPUTS

Description:

The pipeline can be used to pipe one or more objects to a script or function. This directive is used to describe such objects and their types.

If this directive occurs multiple times, each associated help content block is collected in the one documentation entry, in the directives' lexical order.

Examples:

.INPUTS  
None. You cannot pipe objects to Get-Power.

.INPUTS  
For the Value parameter, one or more objects of any kind can be written to the pipeline. However, the object is converted to a string before it is added to the item.

function Process-Thing  
{  
 param ( …  
 [Parameter(ValueFromPipeline=$true)]  
 [object[]]$Value,  
 …  
 )  
…  
}

* + 1. .LINK

Syntax:

.LINK

Description:

This directive specifies the name of a related topic.

If this directive occurs multiple times, each associated help content block is collected in the one documentation entry, in the directives' lexical order.

The Link directive content can also include a URI to an online version of the same help topic. The online version is opens when Get-Help is invoked with the Online parameter. The URI must begin with "http" or "https".

Examples:

.LINK  
Online version: http://www.acmecorp.com/widget.html

.LINK  
Set-ProcedureName

* + 1. .NOTES

Syntax:

.NOTES

Description:

This directive allows additional information about the function or script to be provided. This directive can be used only once in each topic.

Examples:

.Notes  
arbitrary text goes here

* + 1. .OUTPUTS

Syntax:

.OUTPUTS

Description:

This directive is used to describe the objects output by a command.

If this directive occurs multiple times, each associated help content block is collected in the one documentation entry, in the directives' lexical order.

Examples:

.OUTPUTS  
double - Get-Power returns Base to the power Exponent.

.OUTPUTS  
None unless the -PassThru switch parameter is used.

* + 1. .PARAMETER

Syntax:

.PARAMETER <Parameter-Name>

Description:

This directive allows for a detailed description of the given parameter. This directive can be used once for each parameter. Parameter directives can appear in any order in the comment block; however, the order in which their corresponding parameters are actually defined in the source determines the order in which the parameters and their descriptions appear in the resulting documentation.

An alternate format involves placing a parameter description comment immediately before the declaration of the corresponding parameter variable's name. If the source contains both a parameter description comment and a Parameter directive, the description associated with the Parameter directive is used.

Examples:

<#  
 .PARAMETER Base  
 The integer value to be raised to the Exponent-th power.

.PARAMETER Exponent  
 The integer exponent to which Base is to be raised.  
#>

function Get-Power  
{  
 param ([long]$Base, [int]$Exponent)  
 …  
}

function Get-Power  
{  
 param ([long]  
 # The integer value to be raised to the Exponent-th power.  
 $Base,  
 [int]  
 # The integer exponent to which Base is to be raised.  
 $Exponent  
 )  
 …  
}

* + 1. .SYNOPSIS

Syntax:

.SYNOPSIS

Description:

This directive allows for a brief description of the function or script. (The .DESCRIPTION directive (§A.2.1) is intended for a detailed description.) This directive can be used only once in each topic.

Examples:

.SYNOPSIS  
Computes Base to the power Exponent.

1. Grammar

This appendix contains summaries of the lexical and syntactic grammars found in the main document.

* 1. Lexical grammar

input:  
input-elementsopt signature-blockopt

input-elements:  
input-element  
input-elements input-element

input-element:  
whitespace  
comment  
token

signature-block:  
signature-begin signature signature-end

signature-begin:  
new-line-character # SIG # Begin signature block new-line-character

signature:  
base64 encoded signature blob in multiple single-line-comments

signature-end:  
new-line-character # SIG # End signature block new-line-character

* + 1. Line terminators

new-line-character:  
Carriage return character (U+000D)  
Line feed character (U+000A)  
Carriage return character (U+000D) followed by line feed character (U+000A)

new-lines:  
new-line-character  
new-lines new-line-character

* + 1. Comments

comment:  
single-line-comment  
requires-comment  
delimited-comment

single-line-comment:  
# input-charactersopt

input-characters:  
input-character  
input-characters input-character

input-character:  
Any Unicode character except a new-line-character

requires-comment:  
#requires whitespace command-arguments

dash:  
- (U+002D)  
EnDash character (U+2013)  
EmDash character (U+2014)  
Horizontal bar character (U+2015)

dashdash:  
dash dash

delimited-comment:  
<# delimited-comment-textopt hashes >

delimited-comment-text:  
delimited-comment-section  
delimited-comment-text delimited-comment-section

delimited-comment-section:  
>  
hashesopt not-greater-than-or-hash

hashes:  
#  
hashes #

not-greater-than-or-hash:  
Any Unicode character except > or #

* + 1. White space

whitespace:  
Any character with Unicode class Zs, Zl, or Zp  
Horizontal tab character (U+0009)  
Vertical tab character (U+000B)  
Form feed character (U+000C)  
` (The backtick character U+0060) followed by new-line-character

* + 1. Tokens

token:  
keyword  
variable  
command  
command-parameter  
command-argument-token  
integer-literal  
real-literal  
string-literal  
type-literal  
operator-or-punctuator

* + 1. Keywords

keyword: one of  
begin break catch class  
continue data define do  
dynamicparam else elseif end  
exit filter finally for  
foreach from function if  
in param process return  
switch throw trap try  
until using var while

* + 1. Variables

variable:  
$$  
$?  
$^  
$ variable-scopeopt variable-characters  
@ variable-scopeopt variable-characters  
*braced-variable*

*braced-variable:*${ variable-scopeopt braced-variable-characters }

variable-scope:  
global:  
local:  
private:  
script:  
variable-namespace

variable-namespace:  
variable-characters :

variable-characters:  
variable-character  
variable-characters variable-character

variable-character:  
A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nd  
\_ (The underscore character U+005F)  
?

braced-variable-characters:  
braced-variable-character  
braced-variable-characters braced-variable-character

braced-variable-character:  
Any Unicode character except  
 } (The closing curly brace character U+007D)  
 ` (The backtick character U+0060)  
escaped-character

escaped-character:  
` (The backtick character U+0060) followed by any Unicode character

* + 1. Commands

generic-token:  
generic-token-parts

generic-token-parts:  
generic-token-part  
generic-token-parts generic-token-part

generic-token-part:  
expandable-string-literal  
verbatim-here-string-literal  
variable  
generic-token-char

generic-token-char:  
Any Unicode character except  
 { } ( ) ; , | & $  
 ` (The backtick character U+0060)  
 double-quote-character  
 single-quote-character  
 whitespace  
 new-line-character  
escaped-character

generic-token-with-subexpr-start:  
generic-token-parts $(

command-parameter:  
dash first-parameter-char parameter-chars colonopt

first-parameter-char:  
A Unicode character of classes Lu, Ll, Lt, Lm, or Lo  
\_ (The underscore character U+005F)  
?

parameter-chars:  
parameter-char  
parameter-chars parameter-char

parameter-char:  
Any Unicode character except  
 { } ( ) ; , | & . [  
 colon  
 whitespace  
 new-line-character

colon:  
: (The colon character U+003A)

* + 1. Literals

literal:  
integer-literal  
real-literal  
string-literal

Integer Literals

integer-literal:  
decimal-integer-literal  
hexadecimal-integer-literal

decimal-integer-literal:  
decimal-digits numeric-type-suffixopt numeric-multiplieropt

decimal-digits:  
decimal-digit  
decimal-digit decimal-digits

decimal-digit: one of  
0 1 2 3 4 5 6 7 8 9

numeric-type-suffix:  
long-type-suffix  
decimal-type-suffix

hexadecimal-integer-literal:  
0x hexadecimal-digits long-type-suffixopt numeric-multiplieropt

hexadecimal-digits:  
hexadecimal-digit  
hexadecimal-digit decimal-digits

hexadecimal-digit: one of  
0 1 2 3 4 5 6 7 8 9 a b c d e f

long-type-suffix:  
l

numeric-multiplier: one of  
kb mb gb tb pb

Real Literals

real-literal:  
decimal-digits . decimal-digits exponent-partopt decimal-type-suffixopt numeric-multiplieropt  
. decimal-digits exponent-partopt decimal-type-suffixopt numeric-multiplieropt  
decimal-digits exponent-part decimal-type-suffixopt numeric-multiplieropt

exponent-part:  
e signopt decimal-digits

sign: one of  
+  
*dash*

decimal-type-suffix:  
d

String Literals

string-literal:  
expandable-string-literal  
expandable-here-string-literal  
verbatim-string-literal  
verbatim-here-string-literal

expandable-string-literal:  
double-quote-character expandable-string-charactersopt dollarsopt double-quote-character

double-quote-character:  
" (U+0022)  
Left double quotation mark (U+201C)  
Right double quotation mark (U+201D)  
Double low-9 quotation mark (U+201E)

expandable-string-characters:  
expandable-string-part  
expandable-string-characters expandable-string-part

expandable-string-part:  
Any Unicode character except  
 $  
 double-quote-character  
 ` (The backtick character U+0060)  
braced-variable  
$ Any Unicode character except  
 (  
 {  
 double-quote-character  
 ` (The backtick character U+0060)  
$ *escaped-character*  
*escaped-character  
double-quote-character double-quote-character*

*dollars:*$ *dollars* $

expandable-here-string-literal:  
@ double-quote-character whitespaceopt new-line-character  
 expandable-here-string-charactersopt new-line-character double-quote-character @

expandable-here-string-characters:  
expandable-here-string-part  
expandable-here-string-characters expandable-here-string-part

expandable-here-string-part:  
Any Unicode character except  
 $  
 new-line-character  
braced-variable  
$ Any Unicode character except  
 (  
 new-line-character  
$ new-line-character Any Unicode character except double-quote-char  
$ new-line-character double-quote-char Any Unicode character except @  
new-line-character Any Unicode character except double-quote-char  
new-line-character double-quote-char Any Unicode character except @

expandable-string-with-subexpr-start:  
double-quote-character expandable-string-charsopt $(

expandable-string-with-subexpr-end:  
double-quote-char

expandable-here-string-with-subexpr-start:  
@ double-quote-character whitespaceopt new-line-character  
 expandable-here-string-charsopt $(

expandable-here-string-with-subexpr-end:  
new-line-character double-quote-character @

verbatim-string-literal:  
single-quote-character verbatim-string-charactersopt single-quote-char

single-quote-character:  
' (U+0027)  
Left single quotation mark (U+2018)  
Right single quotation mark (U+2019)  
Single low-9 quotation mark (U+201A)  
Single high-reversed-9 quotation mark (U+201B)

verbatim-string-characters:  
verbatim-string-part  
verbatim-string-characters verbatim-string-part

verbatim-string-part:  
Any Unicode character except single-quote-character  
single-quote-character single-quote-character

verbatim-here-string-literal:  
@ single-quote-character whitespaceopt new-line-character  
 verbatim-here-string-charactersopt new-line-character single-quote-character @

verbatim-*here-string-characters:  
verbatim-here-string-part  
verbatim-here-string-characters verbatim-here-string-part*

*verbatim-here-string-part:*Any Unicode character except new-line-character  
new-line-character Any Unicode character except single-quote-character  
new-line-character single-quote-character Any Unicode character except @

* + 1. Simple Names

simple-name:  
simple-name-first-char simple-name-chars

simple-name-first-char:  
A Unicode character of classes Lu, Ll, Lt, Lm, or Lo  
\_ (The underscore character U+005F)

simple-name-chars:  
simple-name-char  
simple-name-chars simple-name-char

simple-name-char:  
A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nd  
\_ (The underscore character U+005F)

* + 1. Type Names

type-name:  
type-identifier  
type-name . type-identifier

type-identifier:  
type-characters

type-characters:  
type-character  
type-characters type-character

type-character:  
A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nd  
\_ (The underscore character U+005F)

array-type-name:  
type-name [

generic-type-name:  
type-name [

* + 1. Operators and punctuators

operator-or-punctuator: one of  
{ } [ ] ( ) @( @{ $( ;  
&& || & | , ++ .. :: .  
! \* / % + 2>&1 1>&2  
dash dash dash  
dash and dash band dash bnot  
dash bor dash bxor dash not  
dash or dash xor  
assignment-operator   
file-redirection-operator  
comparison-operator  
format-operator

assignment-operator: one of  
= dash = += \*= /= %=

file-redirection-operator: one of  
>> > < 2>> 2>

comparison-operator: one of  
dash as dash ccontains dash ceq  
dash cge dash cgt dash cle  
dash clike dash clt dash cmatch  
dash cne dash cnotcontains dash cnotlike  
dash cnotmatch dash contains dash creplace  
dash csplit dash eq dash ge  
dash gt dash icontains dash ieq  
dash ige dash igt dash ile  
dash ilike dash ilt dash imatch  
dash ine dash inotcontains dash inotlike  
dash inotmatch dash ireplace dash is  
dash isnot dash isplit dash join  
dash le dash like dash lt  
dash match dash ne dash notcontains  
dash notlike dash notmatch dash replace  
dash split

format-operator:  
dash f

* 1. Syntactic grammar
     1. Basic concepts

script-file:  
script-block

module-file:  
script-block

interactive-input:  
script-block

data-file:  
statement-list

* + 1. Statements

script-block:  
param-blockopt statement-terminatorsopt script-block-bodyopt

param-block:  
new-linesopt attribute-listopt new-linesopt param new-linesopt  
 ( parameter-listopt new-linesopt )

parameter-list:  
script-parameter  
parameter-list new-linesopt , script-parameter

script-parameter:  
new-linesopt attribute-listopt new-linesopt variable script-parameter-defaultopt

script-parameter-default:  
new-linesopt = new-linesopt expression

script-block-body:  
named-block-list  
statement-list

named-block-list:  
named-block  
named-block-list named-block

named-block:  
block-name statement-block statement-terminatorsopt

block-name: one of  
dynamicparam begin process end

statement-block:  
new-linesopt { statement-listopt new-linesopt }

statement-list:  
statement  
statement-list statement

statement:  
if-statement  
labelopt labeled-statement  
function-statement  
flow-control-statement statement-terminator  
trap-statement  
try-statement  
data-statement  
pipeline statement-terminator

statement-terminator:  
;  
new-line-character

statement-terminators:  
statement-terminator  
statement-terminators statement-terminator

if-statement:  
if new-linesopt ( new-linesopt pipeline new-linesopt ) statement-block  
 elseif-clausesopt else-clauseopt

elseif-clauses:  
elseif-clause  
elseif-clauses elseif-clause

elseif-clause:  
new-linesopt elseif new-linesopt ( new-linesopt pipeline new-linesopt ) statement-block

else-clause:  
new-linesopt else statement-block

labeled-statement:  
switch-statement  
foreach-statement  
for-statement  
while-statement  
do-statement

switch-statement:  
switch new-linesopt switch-parametersopt switch-condition switch-body

switch-parameters:  
switch-parameter  
switch-parameters switch-parameter

switch-parameter:  
-regex  
-wildcard  
-exact  
-casesensitive

switch-condition:  
( new-linesopt pipeline new-linesopt )  
-file new-linesopt switch-filename

switch-filename:  
command-argument  
primary-expression

switch-body:  
new-linesopt { new-linesopt switch-clauses }

switch-clauses:  
switch-clause  
switch-clauses switch-clause

switch-clause:  
switch-clause-condition statement-block statement-terimatorsopt

switch-clause-condition:  
command-argument  
primary-expression

foreach-statement:  
foreach new-linesopt ( new-linesopt variable new-linesopt in new-linesopt pipeline  
 new-linesopt ) statement-block

for-statement:  
for new-linesopt (  
 new-linesopt for-initializeropt statement-terminator  
 new-linesopt for-conditionopt statement-terminator  
 new-linesopt for-iteratoropt  
 new-linesopt ) statement-block  
for new-linesopt (  
 new-linesopt for-initializeropt statement-terminator  
 new-linesopt for-conditionopt  
 new-linesopt ) statement-block  
for new-linesopt (  
 new-linesopt for-initializeropt  
 new-linesopt ) statement-block

for-initializer:  
pipeline

for-condition:  
pipeline

for-iterator:  
pipeline

while-statement:  
while new-linesopt ( new-linesopt while-condition new-linesopt ) statement-block

do-statement:  
do statement-block new-linesopt while new-linesopt ( while-condition new-linesopt )  
do statement-block new-linesopt until new-linesopt ( while-condition new-linesopt )

while-condition:  
new-linesopt pipeline

function-statement:  
function new-linesopt function-name function-parameter-declarationopt { script-block }  
filter new-linesopt function-name function-parameter-declarationopt { script-block }

function-name:  
command-argument

function-parameter-declaration:  
new-linesopt ( parameter-list new-linesopt )

flow-control-statement:  
break label-expressionopt  
continue label-expressionopt  
throw pipelineopt  
return pipelineopt  
exit pipelineopt

label-expression:  
simple-name  
unary-expression

trap-statement:  
trap new-linesopt type-literalopt new-linesopt statement-block

try-statement:  
try statement-block catch-clauses  
try statement-block finally-clause  
try statement-block catch-clauses finally-clause

catch-clauses:  
catch-clause  
catch-clauses catch-clause

catch-clause:  
new-linesopt catch catch-type-listopt statement-block

catch-type-list:  
new-linesopt type-literal  
catch-type-list new-linesopt , new-linesopt type-literal

finally-clause:  
new-linesopt finally statement-block

data-statement:  
data new-linesopt data-name data-commands-allowedopt statement-block

data-name:  
simple-name

data-commands-allowed:  
new-linesopt -supportedcommand data-commands-list

data-commands-list:  
new-linesopt data-command  
data-commands-list , new-linesopt data-command

data-command:  
command-name-expr

pipeline:  
assignment-expression  
expression redirectionsopt pipeline-tailopt  
command pipeline-tailopt

assignment-expression:  
expression assignment-operator statement

pipeline-tail:  
| new-linesopt command  
| new-linesopt command pipeline-tail

command:  
command-name command-elementsopt  
command-invocation-operator command-moduleopt command-name-expr command-elementsopt

command-invocation-operator: one of  
& .

command-module:  
primary-expression

command-name:  
generic-token  
generic-token-with-subexpr

generic-token-with-subexpr:  
No whitespace is allowed between ) and command-name.  
generic-token-with-subexpr-start statement-listopt ) command-name

command-name-expr:  
command-name  
primary-expression

command-elements:  
command-element  
command-elements command-element

command-element:  
command-parameter  
command-argument  
redirection

command-argument:  
command-name-expr

redirections:  
redirection  
redirections redirection

redirection:  
2>&1  
1>&2  
file-redirection-operator redirected-file-name

redirected-file-name:  
command-argument  
primary-expression

* + 1. Expressions

expression:  
logical-expression

logical-expression:  
bitwise-expression  
logical-expression -and new-linesopt bitwise-expression  
logical-expression -or new-linesopt bitwise-expression  
logical-expression -xor new-linesopt bitwise-expression

bitwise-expression:  
comparison-expression  
bitwise-expression -band new-linesopt comparison-expression  
bitwise-expression -bor new-linesopt comparison-expression  
bitwise-expression -bxor new-linesopt comparison-expression

comparison-expression:  
additive-expression  
comparison-expression comparison-operator new-linesopt additive-expression

additive-expression:  
multiplicative-expression  
additive-expression + new-linesopt multiplicative-expression  
additive-expression dash new-linesopt multiplicative-expression

multiplicative-expression:  
format-expression  
multiplicative-expression \* new-linesopt format-expression  
multiplicative-expression / new-linesopt format-expression  
multiplicative-expression % new-linesopt format-expression

format-expression:  
range-expression  
format-expression format-operator new-linesopt range-expression

range-expression:  
array-literal-expression  
range-expression .. new-linesopt array-literal-expression

array-literal-expression:  
unary-expression  
unary-expression , new-linesopt array-literal-expression

unary-expression:  
primary-expression  
expression-with-unary-operator

expression-with-unary-operator:  
, new-linesopt unary-expression  
-not new-linesopt unary-expression  
! new-linesopt unary-expression  
-bnot new-linesopt unary-expression  
+ new-linesopt unary-expression  
dash new-linesopt unary-expression  
pre-increment-expression  
pre-decrement-expression  
cast-expression  
-split new-linesopt unary-expression  
-join new-linesopt unary-expression

pre-increment-expression:  
++ new-linesopt unary-expression

pre-decrement-expression:  
dashdash new-linesopt unary-expression

cast-expression:  
type-literal unary-expression

primary-expression:  
value  
member-access  
element-access  
invocation-expression  
post-increment-expression  
post-decrement-expression

value:  
parenthesized-expression  
sub-expression  
array-expression  
script-block-expression  
hash-literal-expression  
literal  
type-literal  
variable

parenthesized-expression:  
( new-linesopt pipeline new-linesopt )

sub-expression:  
$( new-linesopt statement-listopt new-linesopt )

array-expression:  
@( new-linesopt statement-listopt new-linesopt )

script-block-expression:  
{ new-linesopt script-block new-linesopt }

hash-literal-expression:  
@{ new-linesopt hash-literal-bodyopt new-linesopt }

hash-literal-body:  
hash-entry  
hash-literal-body statement-terminators hash-entry

hash-entry:  
key-expression = new-linesopt statement

key-expression:  
simple-name  
unary-expression

post-increment-expression:  
primary-expression ++

post-decrement-expression:  
primary-expression dashdash

member-access: Note no whitespace is allowed between terms in these productions.  
primary-expression . member-name  
primary-expression :: member-name

element-access: Note no whitespace is allowed between primary-expression and[.  
primary-expression [ new-linesopt expression new-linesopt ]

invocation-expression: Note no whitespace is allowed between terms in these productions.  
primary-expression . member-name argument-list  
primary-expression :: member-name argument-list

argument-list:  
( argument-expression-listopt new-linesopt )

argument-expression-list:  
argument-expression  
argument-expression new-linesopt , argument-expression-list

argument-expression:  
new-linesopt logical-argument-expression

logical-argument-expression:  
bitwise-argument-expression  
logical-argument-expression -and new-linesopt bitwise-argument-expression  
logical-argument-expression -or new-linesopt bitwise-argument-expression  
logical-argument-expression -xor new-linesopt bitwise-argument-expression

bitwise-argument-expression:  
comparison-argument-expression  
bitwise-argument-expression -band new-linesopt comparison-argument-expression  
bitwise-argument-expression -bor new-linesopt comparison-argument-expression  
bitwise-argument-expression -bxor new-linesopt comparison-argument-expression

comparison-argument-expression:  
additive-argument-expression  
comparison-argument-expression comparison-operator  
 new-linesopt additive-argument-expression

additive-argument-expression:  
multiplicative-argument-expression  
additive-argument-expression + new-linesopt multiplicative-argument-expression  
additive-argument-expression dash new-linesopt multiplicative-argument-expression

multiplicative-argument-expression:  
format-argument-expression  
multiplicative-argument-expression \* new-linesopt format-argument-expression  
multiplicative-argument-expression / new-linesopt format-argument-expression  
multiplicative-argument-expression % new-linesopt format-argument-expression

format-argument-expression:  
range-argument-expression  
format-argument-expression format-operator new-linesopt range-argument-expression

range-argument-expression:  
unary-expression  
range-expression .. new-linesopt unary-expression

member-name:  
simple -name  
string-literal  
string-literal-with-subexpression  
expression-with-unary-operator  
value

string-literal-with-subexpression:  
expandable-string-literal-with-subexpr  
expandable-here-string-literal-with-subexpr

expandable-string-literal-with-subexpr:  
expandable-string-with-subexpr-start statement-listopt )  
 expandable-string-with-subexpr-characters expandable-string-with-subexpr-end  
expandable-here-string-with-subexpr-start statement-listopt )  
 expandable-here-string-with-subexpr-characters  
 expandable-here-string-with-subexpr-end

expandable-string-with-subexpr-characters:  
expandable-string-with-subexpr-part  
expandable-string-with-subexpr-characters expandable-string-with-subexpr-part

expandable-string-with-subexpr-part:  
sub-expression  
expandable-string-part

expandable-here-string-with-subexpr-characters:  
expandable-here-string-with-subexpr-part  
expandable-here-string-with-subexpr-characters expandable-here-string-with-subexpr-part

expandable-here-string-with-subexpr-part:  
sub-expression  
expandable-here-string-part

type-literal:  
[ type-spec ]

type-spec:  
array-type-name dimensionopt ]  
generic-type-name generic-type-arguments ]  
type-name

dimension:  
,  
*dimension* ,

generic-type-arguments:  
type-spec  
generic-type-arguments , type-spec

* + 1. Attributes

attribute-list:  
attribute  
attribute-list new-linesopt attribute

attribute:  
[ attribute-name ( attribute-arguments new-linesopt ) new-linesopt ]  
type-literal

attribute-name:  
type-spec

attribute-arguments:  
attribute-argument  
attribute-argument new-linesopt , attribute-arguments

attribute-argument:  
new-linesopt expression  
new-linesopt simple-name = new-linesopt expression

1. References

ANSI/IEEE 754−2008, Binary floating-point arithmetic for microprocessor systems.

ECMA-334, C# Language Specification, 4th edition (June 2006), <http://www.ecma-international.org/publications/standards/Ecma-334.htm>. [This Ecma publication is also approved as ISO/IEC 23270:2006.]

The Open Group Base Specifications: Pattern Matching, IEEE Std 1003.1, 2004 Edition.  
<http://www.opengroup.org/onlinepubs/000095399/utilities/xcu_chap02.html#tag_02_13_01>

The Open Group Base Specifications: Regular Expressions, IEEE Std 1003.1, 2004 Edition.  
<http://www.opengroup.org/onlinepubs/000095399/basedefs/xbd_chap09.html>.

Ecma Technical Report TR/84, Common Language Infrastructure (CLI) - Information Derived from Partition IV XML File, 4th edition (June 2006), <http://www.ecma-international.org/publications/techreports/E-TR-084.htm>. This TR was also published as ISO/IEC TR 23272:2006.

ISO 639-1, Codes for the representation of names of languages -- Part 1: Alpha-2 code.

ISO 3166-1, Codes for the representation of names of countries and their subdivisions -- Part 1: Country codes.

ISO/IEC 10646-1/AMD1:1996, Amendment 1 to ISO/IEC 10646-1:1993, Transformation Format for 16 planes of group 00 (UTF-16).

The Unicode Standard, Edition 5.2. The Unicode Consortium, <http://www.unicode.org/standard/standard.html>.