**PowerShell入門**

URI:

<https://www.tohoho-web.com/ex/power-shell.html>

#### 目次

* [PowerShellとは](https://www.tohoho-web.com/ex/power-shell.html#about)
* [基礎知識](https://www.tohoho-web.com/ex/power-shell.html#basic-know-how)
  + [ポリシー設定](https://www.tohoho-web.com/ex/power-shell.html#policy-settings)
  + [スクリプトファイル実行](https://www.tohoho-web.com/ex/power-shell.html#script-file-execution)
  + [ヘルプ](https://www.tohoho-web.com/ex/power-shell.html#help)
  + [Write-Host と Write-Output](https://www.tohoho-web.com/ex/power-shell.html#write)
  + [コメント(#, <#...#>)](https://www.tohoho-web.com/ex/power-shell.html#comment)
  + [セミコロン(;)](https://www.tohoho-web.com/ex/power-shell.html#semicolon)
  + [継続行(')](https://www.tohoho-web.com/ex/power-shell.html#continuation-line)
  + [Windows PowerShell ISE](https://www.tohoho-web.com/ex/power-shell.html#powershell-ise)
  + [キーワード](https://www.tohoho-web.com/ex/power-shell.html#keywords)
* [変数](https://www.tohoho-web.com/ex/power-shell.html#variables)
  + [変数($xxx)](https://www.tohoho-web.com/ex/power-shell.html#variable)
  + [自動変数](https://www.tohoho-web.com/ex/power-shell.html#automatic-variables)
  + [定数(constant)](https://www.tohoho-web.com/ex/power-shell.html#constant)
  + [DATAセクション(DATA)](https://www.tohoho-web.com/ex/power-shell.html#data)
* [型](https://www.tohoho-web.com/ex/power-shell.html#types)
  + [真偽値($true, $false)](https://www.tohoho-web.com/ex/power-shell.html#boolean)
  + [ヌル型($null)](https://www.tohoho-web.com/ex/power-shell.html#null)
  + [数値](https://www.tohoho-web.com/ex/power-shell.html#number)
  + [文字列](https://www.tohoho-web.com/ex/power-shell.html#string)
  + [配列(@(...))](https://www.tohoho-web.com/ex/power-shell.html#array)
  + [ハッシュテーブル(@{...})](https://www.tohoho-web.com/ex/power-shell.html#hash-table)
  + [列挙型(enum)](https://www.tohoho-web.com/ex/power-shell.html#enum)
  + [オブジェクト](https://www.tohoho-web.com/ex/power-shell.html#object)
  + [スクリプトブロック](https://www.tohoho-web.com/ex/power-shell.html#script-block)
  + [型変換](https://www.tohoho-web.com/ex/power-shell.html#convert)
* [演算子](https://www.tohoho-web.com/ex/power-shell.html#operators)
* [フロー制御](https://www.tohoho-web.com/ex/power-shell.html#flow-control)
  + [if文](https://www.tohoho-web.com/ex/power-shell.html#if)
  + [switch文](https://www.tohoho-web.com/ex/power-shell.html#switch)
  + [for文](https://www.tohoho-web.com/ex/power-shell.html#for)
  + [foreach文](https://www.tohoho-web.com/ex/power-shell.html#foreach)
  + [do-while文](https://www.tohoho-web.com/ex/power-shell.html#do-while)
  + [do-until文](https://www.tohoho-web.com/ex/power-shell.html#do-until)
  + [while文](https://www.tohoho-web.com/ex/power-shell.html#while)
  + [break文](https://www.tohoho-web.com/ex/power-shell.html#break)
  + [continue文](https://www.tohoho-web.com/ex/power-shell.html#continue)
  + [try-catch文](https://www.tohoho-web.com/ex/power-shell.html#try-catch)
  + [trap文](https://www.tohoho-web.com/ex/power-shell.html#trap)
  + [exit文](https://www.tohoho-web.com/ex/power-shell.html#exit)
* [関数](https://www.tohoho-web.com/ex/power-shell.html#functions)
  + [関数定義(function)](https://www.tohoho-web.com/ex/power-shell.html#function)
  + [引数(param)](https://www.tohoho-web.com/ex/power-shell.html#param)
  + [引数の型](https://www.tohoho-web.com/ex/power-shell.html#type-of-arguments)
  + [引数のデフォルト値](https://www.tohoho-web.com/ex/power-shell.html#arguments-default-value)
  + [必須引数(Mandatory)](https://www.tohoho-web.com/ex/power-shell.html#mandatory-arguments)
  + [バリデートセット(ValidateSet)](https://www.tohoho-web.com/ex/power-shell.html#validate-set)
  + [バリデートスクリプト(ValidateScript)](https://www.tohoho-web.com/ex/power-shell.html#validate-script)
  + [可変引数(ValueFromRemainingArguments)](https://www.tohoho-web.com/ex/power-shell.html#varargs)
  + [参照渡し([ref])](https://www.tohoho-web.com/ex/power-shell.html#call-by-reference)
  + [戻り値(return)](https://www.tohoho-web.com/ex/power-shell.html#return)
  + [引数の名前指定](https://www.tohoho-web.com/ex/power-shell.html#arguments-by-name)
* [フィルタリング](https://www.tohoho-web.com/ex/power-shell.html#filtering)
  + [条件抽出(Where/Where-Object)](https://www.tohoho-web.com/ex/power-shell.html#where)
  + [フィルタ(filter)](https://www.tohoho-web.com/ex/power-shell.html#filter)
* [クラス(class)](https://www.tohoho-web.com/ex/power-shell.html#class)
  + [コンストラクタ](https://www.tohoho-web.com/ex/power-shell.html#class-constructor)
  + [スタティックメソッド(static)](https://www.tohoho-web.com/ex/power-shell.html#static)
  + [隠しパラメータ(hidden)](https://www.tohoho-web.com/ex/power-shell.html#hidden)
  + [クラスの継承](https://www.tohoho-web.com/ex/power-shell.html#class-inheritance)
* [スコープ](https://www.tohoho-web.com/ex/power-shell.html#scope)
* [パイプとリダイレクト](https://www.tohoho-web.com/ex/power-shell.html#pipe-and-redirect)
  + [パイプ(|)](https://www.tohoho-web.com/ex/power-shell.html#pipe)
  + [プロセス(process)](https://www.tohoho-web.com/ex/power-shell.html#process)
  + [開始処理・終了処理(begin, end)](https://www.tohoho-web.com/ex/power-shell.html#begin-end)
  + [リダイレクト(>)](https://www.tohoho-web.com/ex/power-shell.html#redirect)
* [コマンドレット](https://www.tohoho-web.com/ex/power-shell.html#cmdlete)
  + [主なコマンドレット](https://www.tohoho-web.com/ex/power-shell.html#common-cmdlet)
  + [Linuxコマンドに対応するコマンドレット](https://www.tohoho-web.com/ex/power-shell.html#linux-vs-power-shell)
* [その他](https://www.tohoho-web.com/ex/power-shell.html#others)
  + [正規表現](https://www.tohoho-web.com/ex/power-shell.html#regexp)
  + [フォーマット](https://www.tohoho-web.com/ex/power-shell.html#format)
  + [ファイルを読み込む](https://www.tohoho-web.com/ex/power-shell.html#read-file)
  + [コンソールから読み込む](https://www.tohoho-web.com/ex/power-shell.html#read-from-console)
  + [一時停止(Pause)](https://www.tohoho-web.com/ex/power-shell.html#pause)
  + [ストリクトモード](https://www.tohoho-web.com/ex/power-shell.html#strict-mode)
  + [プロンプト(prompt)](https://www.tohoho-web.com/ex/power-shell.html#prompt)
  + [プロファイル($profile)](https://www.tohoho-web.com/ex/power-shell.html#profile)
  + [コマンド置換($(...))](https://www.tohoho-web.com/ex/power-shell.html#command-replace)
* [リンク](https://www.tohoho-web.com/ex/power-shell.html#links)

#### PowerShellとは

* Microsoft が開発したシェルであり、スクリプト言語です。
* Windows 用に開発されましたが、中核である PowerShell Core は Linux や macOS にも移植されています。
* 拡張子は \*.ps1 です。
* Linux のパイプラインは文字列情報を引き渡すのに対し、PowerShell ではオブジェクト情報を引き渡すことができます。
* 変数名、関数名等の大文字・小文字は無視されます。
* 他言語ではエスケープにバックスラッシュ(\)を使うことが多いですが、PowerShell ではバッククォート(`)を使用します。
* 環境変数やレジストリをドライブとして参照することができます。

ここではある程度他の言語を習得している方を対象に、PowerShell の概要について簡単に説明していきます。バージョンは 7.4.0 を対象としています。

#### 基礎知識

##### ポリシー設定

Windows のデフォルト設定では管理者権限が無いと PowerShell を実行できない状態になっています。下記を実行してポリシー設定を行ってください。RemoteSigned は、ローカルフォルダで作成したスクリプトは実行できますが、インターネットを経由して入手したスクリプトを実行するには署名が必要であることを示します。

**Get-ExecutionPolicy** # 現在のポリシーを表示

Resricted

**Set-ExecutionPolicy RemoteSigned** # ポリシーを RemoteSigned に設定

##### スクリプトファイル実行

スクリプトファイルは拡張子が .ps1 のファイルに記述します。スクリプトファイルは下記の様にして実行します。

PS C:\Work> **.\test.ps1**

##### ヘルプ

Get-Help はコマンドレットに関するヘルプを表示します。

**Get-Help** Write-Host

-online をつけるとブラウザでコマンドレットの詳細情報を表示することができます。

Get-Help Write-Host **-online**

##### Write-Host と Write-Output

Write-Host は文字列をコンソールに出力します。Write-Output はオブジェクトを Standard output Stream に出力します。

**Write-Host** "Hello world!" # 文字列をコンソールに出力する。

**Write-Object** "Hello world!" # オブジェクト情報を出力する。

Write-Host は必ずコンソールに出力されますが、Write-Output の出力はパイプ(|)で次のコマンドに渡すことができます。

**Write-Host** "C:\" | Get-ChildItem # C:\ がコンソールに出力されるのみ

**Write-Output** "C:\" | Get-ChildItem # C:\ を対象に Get-ChildItem が実行される

Write-Host に **-NoNewLine** オプションをつけると改行無しで出力します。

Write-Host **-NoNewLine** "Hello "

Write-Host **-NoNewLine** "world!"

##### コメント(#, <#...#>)

**#** から行末まではコメントとみなされます。

**# この行はコメントです**

Write-Host "Hello world!" **# この部分もコメントです**

**<#** ... **#>** は複数行のコメントを記述します。

**<#** コメント1

コメント2

コメント3 **#>**

##### セミコロン(;)

セミコロン(**;**) を用いて1行に複数の文を記述することができます。

$a = "Hello" **;** Write-Host $a

##### 継続行(')

行末にバッククォート(`)を書くことで、行末の改行文字を無効化し、ひとつの文を複数行に分けて記述することができます。

Get-ChildItem **`**

-Attribute Directory **`**

-Exclude .\*

##### Windows PowerShell ISE

PowerShell の開発環境として Windows PowerShell ISE が提供されています。[スタート] から Windows PowerShell ISE を検索して起動します。

Windows PowerShell ISE

デフォルトでは左上段にスクリプトウィンドウ、左下段に実行ウィンドウ、右側にコマンドウィンドウが表示されます。コマンドウィンドウでは各コマンドレットのヘルプや引数入力フォームを表示することができます。スクリプトウィンドウに入力したスクリプトは実行ボタン(▶)を押すことで実行できます。

|  |  |
| --- | --- |
| メニューバー | |
| スクリプトウィンドウ | コマンド ウィンドウ |
| 実行ウィンドウ |

ブレークポイントを設定してスクリプトをデバッグすることもできます。デバッガを使用するにはスクリプトをファイルとして保存し、下記の様にポリシー変更する必要があります。

Set-ExecutionPolicy -Scope CurrentUser

ExecutionPolicy: RemoteSigned

##### キーワード

言語としては下記のキーワードが定義されています。

* [begin](https://www.tohoho-web.com/ex/power-shell.html#begin-end)
* [break](https://www.tohoho-web.com/ex/power-shell.html#break)
* [catch](https://www.tohoho-web.com/ex/power-shell.html#try-catch)
* [class](https://www.tohoho-web.com/ex/power-shell.html#class)
* [continue](https://www.tohoho-web.com/ex/power-shell.html#continue)
* [data](https://www.tohoho-web.com/ex/power-shell.html#data)
* define(予約)
* [do](https://www.tohoho-web.com/ex/power-shell.html#do-while)
* dynamicparam
* [else](https://www.tohoho-web.com/ex/power-shell.html#if)
* [elseif](https://www.tohoho-web.com/ex/power-shell.html#if)
* [end](https://www.tohoho-web.com/ex/power-shell.html#begin-end)
* [enum](https://www.tohoho-web.com/ex/power-shell.html#enum)
* [exit](https://www.tohoho-web.com/ex/power-shell.html#exit)
* [filter](https://www.tohoho-web.com/ex/power-shell.html#filter)
* [finally](https://www.tohoho-web.com/ex/power-shell.html#try-catch)
* [for](https://www.tohoho-web.com/ex/power-shell.html#for)
* [foreach](https://www.tohoho-web.com/ex/power-shell.html#foreach)
* from(予約)
* [function](https://www.tohoho-web.com/ex/power-shell.html#function)
* [hidden](https://www.tohoho-web.com/ex/power-shell.html#hidden)
* [if](https://www.tohoho-web.com/ex/power-shell.html#if)
* [in](https://www.tohoho-web.com/ex/power-shell.html#foreach)
* [param](https://www.tohoho-web.com/ex/power-shell.html#param)
* [process](https://www.tohoho-web.com/ex/power-shell.html#process)
* [return](https://www.tohoho-web.com/ex/power-shell.html#return)
* [static](https://www.tohoho-web.com/ex/power-shell.html#static)
* [switch](https://www.tohoho-web.com/ex/power-shell.html#switch)
* [throw](https://www.tohoho-web.com/ex/power-shell.html#try-catch)
* [trap](https://www.tohoho-web.com/ex/power-shell.html#trap)
* [try](https://www.tohoho-web.com/ex/power-shell.html#try-catch)
* [until](https://www.tohoho-web.com/ex/power-shell.html#do-until)
* [using](https://www.tohoho-web.com/ex/power-shell.html#scope)
* var(予約)
* [while](https://www.tohoho-web.com/ex/power-shell.html#while)

#### 変数

##### 変数($xxx)

変数は $変数名 で表します。

**$Name** = "Yamada"

Write-Output "My name is **$Name**."

変数の値を削除するには Clear-Variable を使用するか、値 $null を設定します。変数を削除するには Remove-Variable または Remove-Item を使用します。

**Clear-Variable** -Name a # 値をクリア

$a = **$null** # 値をクリア

**Remove-Variable** -Name a # 変数を削除

**Remove-Item** -Path Variable:\a # 変数を削除

##### 自動変数

デフォルトで下記などの自動変数が定義されています。

**$true** # 真値

**$false** # 偽値

**$null** # ヌル値

**$$** # 最後に実行したステートメント

**$?** # 最後に実行したステートメントの結果(True/False)

**$LastExitCode** # 最後にエラーを起こしたコマンドの終了コード

**$\_** # パイプライン処理中の対象オブジェクト

**$this** # クラスインスタンス

**$Error** # 最後にエラーとなったステートメントのエラー情報

**$args** # 関数に渡された引数リスト

**$Host** # ホスト情報

**$Home** # ユーザのホームディレクトリ

**$PSHome** # PowerShellがインストールされているディレクトリ

**$profile** # プロファイルファイルのパス名

**$PWD** # カレントディレクトリのパス名

**$PSVersionTable** # PowerShellのバージョン情報

**$IsWindows** # 環境が Windows であるか否か

**$IsLinux** # 環境が Linux であるか否か

**$IsMacOS** # 環境が Mac OS であるか否か

詳細は下記を参照してください。

* [about\_Automatic\_Variables](https://learn.microsoft.com/ja-jp/powershell/module/microsoft.powershell.core/about/about_automatic_variables?view=powershell-7.2) (Microsoft)

##### 定数(constant)

定数を定義するには **Set-Variable** の **Constant** オプションを使用します。定数は通常すべて大文字で定義します。

**Set-Variable -Name** MAX\_SIZE **-Value** 256 **-Option Constant**

Write-Host $MAX\_SIZE # 256

$MAX\_SIZE = 512 # 定数を変更しようとするとERROR

##### DATAセクション(DATA)

DATA セクションでは **ConvertFrom-StringData** を用いて文字列をハッシュテーブルに変換し、キーと値のリストを定義します。コマンドレットの国際化対応などで利用されています。

$ERROR\_MSG = **DATA {**

**ConvertFrom-StringData** @'

ERROR = エラー

SYNTAX\_ERROR = シンタックスエラー

PERMISSION\_DENIED = 権限がありません

'@

**}**

$ERROR\_MSG.SYNTAX\_ERROR # シンタックスエラー

#### 型

整数、実数、文字列、配列、ハッシュテーブルなどを使用できます。

$a = 123 # 整数

$a = 0x00FF # 16進数

$a = 0b11110000 # 2進数

$a = 1.23 # 実数

$a = 1.23e3 # 実数(1.23×10の3乗)

$a = "ABC" # 文字列(変数展開あり)

$a = 'ABC' # 文字列(変数展開なし)

$a = @("A", "B", "C") # 配列

$a = 1..3 # @(1, 2, 3)と同じ

$a = "A", "B", "C" # @("A", "B", "C")と同じ

$a = @{"A"=1; "B"=2; "C"=3} # ハッシュテーブル

下記の様に変数の型を明示することができます。

**[string]**$str = 123 # 文字列の "123" が設定される

Write-Host ($str + 456) # 文字列連結で "123456" が表示される

**[int]**$num = 123 # 整数に限定

$num = "ABC" # 文字列を代入しようとするとエラー

指定可能な型には下記などがあります。

---- 真偽 ----

bool System.Boolean ブーリアン($true/$false)

---- 文字・文字列 ----

char System.Char 文字("A", 'A')

string System.String 文字列("ABC", 'ABC')

---- 正数 ----

byte System.Byte 8ビット整数(0～255)

sbyte System.SByte 8ビット整数(-128～127)

short System.Int16 16ビット整数(-32,768～32,767)

ushort System.UInt16 16ビット整数(0～65,535)

int16 System.Int16 16ビット整数(-32,768～32,767)

uint16 System.UInt16 16ビット整数(0～65,535)

int System.Int32 32ビット整数(-2,147,483,648～2,147,483,647)

uint System.UInt32 32ビット整数(0～4,294,967,295)

int32 System.Int32 32ビット整数(-2,147,483,648～2,147,483,647)

uint32 System.UInt32 32ビット整数(0～4,294,967,295)

long System.Int64 64ビット整数(-9,223,372,036,854,775,808～9,223,372,036,854,775,807)

ulong System.UInt64 64ビット整数(0～18,446,744,073,709,551,615)

int64 System.Int64 64ビット整数(-9,223,372,036,854,775,808～9,223,372,036,854,775,807)

uint64 System.UInt64 64ビット整数(0～18,446,744,073,709,551,615)

bigint System.BigInteger 制限なし？

---- 実数 ----

single System.Single 32ビット実数(-3.402823E+38～3.402823E+38)

float System.Single 32ビット実数(-3.402823E+38～3.402823E+38)

double System.Double 64ビット実数(-1.79769313486232E+308～1.79769313486232E+308)

---- 10進数 ----

decimal System.Decimal 10進数(-79228162514264337593543950335～79228162514264337593543950335)

---- その他 ----

void System.Void 型無し

array System.Array 配列

datetime System.DateTime 日時

timespan System.TimeSpan タイムスパン

type System.Type 型

hashtable System.Collections.Hashtable ハッシュテーブル

psobject System.Management.Automation.PSObject PowerScriptオブジェクト

Regex System.Text.RegularExpressions.Regex 正規表現

scriptblock System.Management.Automation.ScriptBlock スクリプトブロック

switch System.Management.Automation.SwitchParameter switchパラメータ

xml System.Xml.XmlDocument XMLドキュメント

型エイリアスの一覧は下記で確認することができます。

[PSObject].Assembly.GetType("System.Management.Automation.TypeAccelerators")::Get.GetEnumerator()

**GetType()** で型を調べることができます。

$a = 123

Write-Host $a.**GetType()** # System.Int32

##### 真偽値($true, $false)

**bool** 型は真偽値を定義します。真は **$true**、偽は **$false** で示します。if 文などの条件式においては、**$false**、数値の 0 や 0.0、空文字列、空配列などは $false と見なされます。

**[bool]**$flag = **$true**

##### ヌル型($null)

**$null** は値無しを示す特別な値を示します。

$a = **$null**

##### 数値

整数を下記の様に代入すると Int32, Int64, Decimal のいずれかで定義されます。

$a = 123 # Int32

$a = 1234567890 # Int64

$a = 12345678901234567890 # Decimal

**0x** で始まる数字は16進数を示します。

$a = **0x**77FF # 16進数

末尾にサフィックスをつけることにより型や乗数を指定することができます。

$a = 123**y** # SByte

$a = 123**uy** # Byte

$a = 123**s** # Int16

$a = 123**us** # UInt16

$a = 123**u** # UInt32

$a = 123**l** # Int32

$a = 123**ul** # UInt32

$a = 123**n** # BigInteger

$a = 123**d** # Decimal

$a = 123**kb** # KB(123×1024)

$a = 123**mb** # MB(123×1024×1024)

$a = 123**gb** # GB(123×1024×1024×1024)

$a = 123**tb** # TB(123×1024×1024×1024×1024)

$a = 123**pb** # PB(123×1024×1024×1024×1024×1024)

実数は Double で定義されます。1.23e4 は 1.23 × 104 を意味します。

$a = 1.23 # Double

$a = 1.23e4 # Double

実数にもサフィックスをつけることができます。

$a = 1.23**d** # Decimal

$a = 1.23**kb** # KB(1.23×1024)

$a = 1.23**mb** # MB(1.23×1024×1024)

$a = 1.23**gb** # GB(1.23×1024×1024×1024)

$a = 1.23**tb** # TB(1.23×1024×1024×1024×1024)

$a = 1.23**pb** # PB(1.23×1024×1024×1024×1024×1024)

数値に関して下記などの関数が用意されています。**Round()** は通常の四捨五入ではなく、小数部が .5 の時、整数部が奇数であれば繰り上げ、偶数であれば繰り下げを行う **銀行丸め** を行うので注意してください。

$a = **[Math]::Floor(**123.4**)** # 小数点切り捨て

$a = **[Math]::Ceiling(**123.4**)** # 小数点切り上げ

$a = **[Math]::Round(**123.4**)** # 四捨五入(銀号丸め)

$a = **[Math]::Abs(**-123**)** # 絶対値

$a = **[Math]::Pow(**2, 10**)** # 累乗(2の10乗)

$a = **[Math]::Max(**3, 5**)** # 大きい方を選択

$a = **[Math]::Min(**3, 5**)** # 小さい方を選択

$a = **[Math]::Sin(**[Math]::PI / 2**)** # 三角関数(sin)

##### 文字列

文字列はダブルクォート(")またはシングルクォート(')で囲みます。

$a = **"ABC"**

$a = **'ABC'**

"..." や '...' の中でクォーテーションを記述するには次の様にします。

$a = "AAA**'**AAA" # "..." の中では ' を使用可能

$a = 'AAA**"**AAA' # '...' の中では " を使用可能

$a = "AAA**""**AAA" # "..." の中では "" の様に二重に書くと " となる

$a = 'AAA**''**AAA' # '...' の中では '' の様に二重に ' となる

$a = "AAA**`"**AAA" # "..." の中では ` でエスケープして `" と書くと " となる

ダブルクォート(")で囲んだ場合は変数の展開が行われます。

$name = "Yamada"

Write-Host "Hello **$name**!"

変数名と後続の文字の境界があいまいな場合は変数名を {...} で囲みます。

$size = 123

Write-Host "File size: **${size}**KB"

@"..."@ や @'...'@ で囲むことにより改行を含む複数行の文字列を表現できます。ヒアドキュメントと呼ばれます。" や ' もそのまま使用することができます。@"..."@ の中では変数が展開されます。

$a = **@"**

This is Japan.

That is America.

**"@**

文字列には下記などのメソッドを使用することができます。

*str*.**SubString(***n*, *m***)** # *n*番目から*m*文字分の部分文字列を得る

*str*.**Remove(***n*, *m***)** # *n*番目から*m*文字分の部分文字列を削除する

*str*.**Insert(***n*, *str1***)** # *n*番目の位置に*str1*を挿入する

*str*.**Replace(***str1***,** *str2***)** # *str1*を*str2*に置換する

*str*.**IndexOf(***str1***)** # *str1*が出現する位置を返す。見つからなければ-1

*str*.**LastIndexOf(***str1***)** # *str1*が最後から見て出現する位置を返す。見つからなければ-1

*str*.**Split(***str1***)** # *str1*で分割した配列を返す

[string]::**Join(***str*, *arr***)** # 文字列配列*arr*を区切り文字*str1*で連結する

*str*.**Contains(***str1***)** # *str1*を含んでいるか調べる

*str*.**StartsWith(***str1***)** # *str1*で始まっているか調べる

*str*.**EndsWith(***str1***)** # *str1*で終わっているか調べる

*str*.**ToUpper()** # 大文字に変換する

*str*.**ToLower()** # 小文字に変換する

*str*.**Trim()** # 前後の空白を削除する

*str*.**TrimStart()** # 前の空白を削除する

*str*.**TrimEnd()** # 後の空白を削除する

文字列の置換には .Replace() と -Replace() があります。.Replace() は大文字・小文字を区別し、正規表現は使用できません。-Replace() は大文字・小文字を区別せず、正規表現を使用できます。

"ABCDEFG"**.Replace(**"cde", "XXX"**)** # 大文字・小文字を区別するのは置換されない

"ABCDEFG"**-Replace(**"cde", "XXX"**)** # 大文字・小文字を区別しないので置換される

"ABCDEFG"**.Replace(**"C.E", "XXX"**)** # 正規表現は使用できないので置換されない

"ABCDEFG"**-Replace(**"C.E", "XXX"**)** # 正規表現を使用できるので置換される

"..." 文字列の中では下記のエスケープシーケンスを利用できます。他言語の様なバックスラッシュ(\)ではなく、バッククォート(`)を用います。改行コードを Windows に合わせた CRLF とする場合は `r`n とします。

**`0** ヌル値(NUL)

**`a** アラート(BEL)

**`b** バックスペース(BS)

**`e** エスケープ(ESC)

**`f** フォームフィード(FF)

**`n** 改行(LF)

**`r** キャリッジリターン(CR)

**`t** 水平タブ(TAB)

**`v** 垂直タブ(HT)

**`'** シングルクォート(')

**@"** ダブルクォート(")

**``** バッククォート(`)

**@$** ドルマーク($)

**`u{x}** Unicode文字(xは16進)

##### 配列(@(...))

配列は下記の様に定義します。

$a = **1, 2, 3**

$a = **"A", "B", "C"**

**@(**...**)** で囲むことにより配列であることを明確に定義することもできます。

$a = **@(**1, 2, 3**)**

$a = **@(**"A", "B", "C"**)**

**..** 演算子を使用して、配列の範囲を指定することもできます。

$a = 1**..**3 # @(1, 2, 3) と同義

$a = "A"**..**"C" # @("A", "B", "C") と同義

配列要素には $変数名[*n*] でアクセスできます。*n* は 0 から始まります。

$a = @("A", "B", "C")

Write-Host **$a[0]** # "A"

Write-Host **$a[1]** # "B"

Write-Host **$a[2]** # "C"

$変数名[*n*..*m*] はスライスと呼ばれる配列の部分要素を取り出します。

$a = @("A", "B", "C", "D", "E")

$b = **$a[1..3]** # @("B", "C", "D")

配列に対して + 演算子で要素や配列を追加することができます。

$a = @("A", "B", "C")

$b = $a **+** "D" # @("A", "B", "C", "D")

$c = $b **+** @("E", "F") # @("A", "B", "C", "D", "E", "F")

配列の個数は Length や Count で取得できます。

$a = @("A", "B", "C")

Write-Host $a.**Length** # 3

Write-Host $a.**Count** # 3

配列をループ処理するには for() や foreach() を用います。

$a = @("A", "B", "C")

**for (**$i = 0**;** $i -lt $a.Length**;** $i++**) {**

Write-Host $a[$i]

**}**

**foreach (**$item **in** $a**) {**

Write-Host $item

**}**

配列に比較演算子を用いると条件に合致した要素のみを取り出すことができます。

$a = @(1, 2, 3, 4, 5)

$a **-gt** 3 # 4, 5

##### ハッシュテーブル(@{...})

ハッシュテーブルは下記の様に定義します。

$a = **@{"A"=1; "B"=2; "C"=3}**

$a = **@{**

"A" = 1

"B" = 2

"C" = 3

**}**

$変数名[*key*] で各要素を参照・設定できます。

$a = @{"A"=1; "B"=2; "C"=3}

Write-Host **$a["B"]** # 2

**$a["C"]** = 4 # 4

要素を追加するには Add()、削除するには Remove() を用います。

$a = @{"A"=1; "B"=2; "C"=3}

$a.**Add(**"D", 4**)**

$a.**Remove(**"D"**)**

ハッシュテーブルが指定したキーの要素を持っているかを判断するには ContainsKey() を、指定した値の要素を持っているかを判断するには ContainsValue() を用います。

$a = @{"A"=1; "B"=2; "C"=3}

$a.**ContainsKey(**"B"**)** # True

$a.**ContainsKey(**"D"**)** # False

$a.**ContainsValue(**2**)** # True

$a.**ContainsValue(**4**)** # False

ハッシュテーブルをループで処理するには foreach() を用います。

$a = @{"A"=1; "B"=2; "C"=3}

**foreach (**$key **in** $a.Keys**) {**

Write-Host $key "=" $a[$key]

**}**

##### 列挙型(enum)

**enum** は列挙型を定義します。

**enum** Color **{**

red

green

blue

**}**

$color = [Color]::red

##### オブジェクト

例えば Get-History コマンドレットを実行すると下記の様に表示されます。

PS C:\Work> **Get-History**

Id Duration CommandLine

-- -------- -----------

1 0.009 Write-Host "Hello Yamada!"

2 0.003 Write-Host "Hello Suzuki!"

3 0.006 Write-Host "Hello Tanaka!"

PS C:\Work>

しかし、Get-History は上記よりも多くの情報を持つオブジェクト情報を出力しています。オブジェクト情報の詳細を見るには出力を ConvertTo-Json で JSON 形式に変換するとわかります。

PS C:\Work> Get-History | **ConvertTo-Json**

[

{

"Id": 1,

"CommandLine": "Write-Host \"Hello Yamada!\"",

"ExecutionStatus": 4,

"StartExecutionTime": "2023-11-21T22:08:59.132628+09:00",

"EndExecutionTime": "2023-11-21T22:08:59.1425216+09:00",

"Duration": {

"Ticks": 98936,

"Days": 0,

"Hours": 0,

:

下記の様に Get-History の出力を変数に代入することで、オブジェクト情報の詳細にアクセスすることができます。

PS C:\Work> **$a = Get-History**

PS C:\Work> **$a[0].CommandLine**

Write-Host "Hello Yamada!"

PS C:\Work> **$a[0].Duration.Ticks**

98936

PS C:\Work>

##### スクリプトブロック

**{**...**}** をスクリプトブロックとして変数や引数に指定することができます。スクリプトブロックを実行するには **&** を使用します。

$square = **{** param($x); $x \* $x **}**

**&** $square 5 # 25

##### 型変換

下記の様に型変換することができます。

$a = [int]"255" # 10進文字列 → Int32

$a = [int64]"255" # 10進文字列 → Int64

$a = [string]255 # Int32 → 10進文字列

$a = [int]3.4 # Double → Int32

$a = [double]3 # Int32 → Double

$a = [System.Convert]::ToInt32("FF", 16) # 16進文字列 → Int32

$a = (255).ToString("x") # Int32 → 16進文字列(小文字)

$a = (255).ToString("X") # Int32 → 16進文字列(大文字)

$a = [System.Convert]::ToInt32("1111", 2) # 2進文字列 → Int32

$a = [convert]::ToString(255, 2) # Int32 → 2進文字列

$a = [datetime]"2023/12/31 12:59:59" # 文字列 → DateTime

$a = ($date).ToString("yyyy/MM/dd HH:mm:ss") # DateTime → 文字列

#### 演算子

算術演算子には下記があります。

5 **+** 3 # 加算

5 **-** 3 # 減算

5 **\*** 3 # 乗算

5 **/** 3 # 除算

5 **%** 3 # 剰余

+ 演算子は配列はハッシュテーブルにも使用することができます。

Write-Output (@("A", "B") **+** @("C", "D")) # @("A", "B", "C", "D")

Write-Output (@{"A"=1; "B"=2} **+** @{"C"=3; "D"=4}) # @{"A"=1; "B"=2; "C"=3; "D"=4}

\* 演算子は配列や文字列にも使用することができます。

Write-Output (@("A", "B", "C") **\*** 2) # @("A", "B", "C", "A", "B", "C")

Write-Output ("ABC" **\*** 2) # "ABCABC"

代入演算子には下記があります。

$a **=** 3 # 代入

$a **+=** 3 # $a = $a + 3

$a **-=** 3 # $a = $a - 3

$a **\*=** 3 # $a = $a \* 3

$a **/=** 3 # $a = $a / 3

$a **%=** 3 # $a = $a % 3

単項演算子には下記があります。

$a**++** # $a = $a + 1 (加算前の値が参照される)

**++**$a # $a = $a + 1 (加算後の値が参照される)

$a**--** # $a = $a - 1 (減算前の値が参照される)

**--**$a # $a = $a - 1 (減算後の値が参照される)

比較演算子には下記があります。

$a -**eq** $b # 等しければ(EQual)

$a **-ne** $b # 等しくなければ(Not Equal)

$a **-gt** $b # $a が $b より大きければ(Greater Than)

$a **-lt** $b # $a が $b より小さければ(Less Than)

$a **-ge** $b # $a が $b 以上であれば(Greater than or Equal)

$a **-le** $b # $a が $b 以下であれば(Less than or Equal)

正規表現やワイルドカードで比較することもできます。

$a **-match** $b # 正規表現にマッチすれば(大文字小文字無視)

$a **-notmatch** $b # 正規表現にマッチしなければ(大文字小文字無視)

$a **-like** $b # ワイルドカードにマッチすれば(大文字小文字無視)

$a **-notlike** $b # ワイルドカードにマッチしなければ(大文字小文字無視)

束縛演算子には下記があります。

$arr -contains $a # 配列 $arr が $a を含んでいれば真

$arr -nocontains $a # 配列 $arr が $a を含んでいなければ真

文字列を比較する場合、上記は大文字・小文字を区別しません。i をつけると明示的に区別しない、c をつけると区別するようになります。

**-eq -ieq -ceq**

**-ne -ine -cne**

**-gt -igt -cgt**

**-lt -ilt -clt**

**-ge -ige -cge**

**-le -ile -cle**

**-match -imatch -cmatch**

**-notmatch -inotmatch -cnotmatch**

**-like -iline -clike**

**-notlike -inotlike -cnotlike**

**-contains -icontains -ccontains**

**-nocontains -inocontains -cnocontains**

論理演算子には下記があります。

$a **-and** $b # $a かつ $b が真(AND)

$a **-or** $b # $a または $b が真(OR)

**-not** $a # $a が真ではない(NOT)

ビット演算子には下記があります。

$a = 0b1100 **-band** 0b0110 # $a AND $b 0b0100

$a = 0b1100 **-bor** 0b0110 # $a OR $b = 0b1111

$a = **-bnot** 0b0110 # NOT $a = 0x1001

分割演算子・結合演算子には下記があります。

Write-Output ("Red,Green,Blue" **-split** ",") # "Red", "Green", "Blue"

Write-Output (**-join** "Red", "Green", "Blue") # "RedGreenBlue"

Write-Output ("Red", "Green", "Blue" **-join** ",") # "Red,Green,Blue"

型演算子には下記があります。

"ABC" **-is** [string] # "ABC"が文字列であれば真

"ABC" **-isnot** [string] # "ABC"が文字列で無ければ真

123 **-as** [string] # 123 を文字列に変換

#### フロー制御

##### if文

**if** (*expr1*) { *statement1* } **elseif** (*expr2*) { *statement2* } **else** { *statement3* } は、もし(if) *expr1* が真であれば *statement1* を、さもなくばもし(elseif) *expr2* が真の場合は *statement2* を、さもなくば(else) *statement3* を実行します。elseif は 0個以上、else は 0個または 1個記述できます。

$a = 5

**if (**$a -eq 5**) {**

Write-Output "等しい"

**} elseif (**$a -lt 5**) {**

Write-Output "小さい"

**} else {**

Write-Output "大きい"

**}**

##### switch文

**switch** (*expr*) { *val1* { *statement1* }... } は、*expr* の値が *val1* であれば *statement1* を実行します。

**switch (**2**) {**

1 { Write-Output "One" }

2 { Write-Output "Two" }

3 { Write-Output "Three" }

Default { Write-Output "Other" }

**}**

**-Wildcard** を指定するとワイルドカードを指定できます。

switch **-Wildcard** ("file.txt") {

"\*.txt" { Write-Host "Text" }

"\*.html" { Write-Host "HTML" }

"\*.png" { Write-Host "PNG" }

}

**-Regex** を指定すると正規表現を指定できます。

switch **-Regex** ("file.txt") {

"\.txt$" { Write-Host "Text" }

"\.html$" { Write-Host "HTML" }

"\.png$" { Write-Host "PNG" }

}

下記の様に条件式を記述することもできます。

switch ("file.txt") {

{**$\_ -match** "\.txt$"} { Write-Host "Text" }

{**$\_ -match** "\.html$"} { Write-Host "HTML" }

{**$\_ -match** "\.png$"} { Write-Host "PNG" }

}

##### for文

**for** (*expr1*; *expr2*; *expr3*) { *statements* } は、*expr1* を実行し、*statements* と *expr3* を繰り返し実行しながら、*expr2* が偽になるとループを終了します。下記の例では $i に 1 を代入し、$i の値をインクリメントしながら $i が 10 以下である間、"Count $i" を出力します。

**for (**$i = 1**;** $i -le 10**;** $i++**) {**

Write-Output "Count $i"

**}**

##### foreach文

**foreach** (*var* **in** *collection*) { *statement* } は、*collection* の各要素を *var* に代入しながら *statement* を実行します。

**foreach (**$color **in** "Red", "Green", "Blue"**) {**

Write-Host $color

**}**

**foreach** は **ForEach-Object** のエイリアスです。下記の様にオブジェクトリストをパイプで受け取り、個々のオブジェクトに対して処理を行うこともできます。

Get-Process | **foreach** {

Write-Output ($\_.Name + " : " + $\_.CPU)

}

**%** もまた **ForEach-Object** のエイリアスです。

Get-Process | **%**{ $\_.Name }

BEGIN ブロックや END ブロックを記述することもできます。

Get-Process | %**{ Write-Host "BEGIN" }**{ $\_.Name }**{ Write-Host "END" }**

##### do-while文

**do** { *statements* } **while** (*condition*) は、*statements* を実行し、*condition* が真である間ループを繰り返します。

$i = 1;

**do {**

Write-Output "Count $i"

**} while (**++$i -le 10**)**

##### do-until文

**do** { *statements* } **until** (*condition*) は、*statements* を実行し、*condition* が真になるまでループを繰り返します。

$i = 1;

**do {**

Write-Output "Count $i"

**} until (**++$i -gt 10**)**

##### while文

**whilte** (*condition*) { *statements* } は、*condition* が真の間 *statements* を繰り返し実行します。

$i = 0;

**while (**$i++ -lt 10**) {**

Write-Output "Count $i"

**}**

##### break文

**break** は一番内側のループを抜けます。

for ($i = 1; $i -le 10; $i++) {

Write-Output "Count $i"

if ($i -eq 5) {

**break**

}

}

[switch](https://www.tohoho-web.com/ex/power-shell.html#switch)文の中で使用すると、次の条件を判断せずに switch文を終了します。

switch (10) {

{$\_ -gt 3 } { Write-Host "Larger than 3"; **break** } # 以降の条件は判断せずに終了する

{$\_ -gt 5 } { Write-Host "Larger than 5"; **break** }

}

##### continue文

**continue** は一番内側のループの残りの処理をスキップして次のループに移ります。

for ($i = 1; $i -le 5; $i++) {

if ($i -eq 3) {

**continue**

}

Write-Output $i # 1, 2, 4, 5

}

##### try-catch文

他の言語と同様 **try**, **catch**, **finally**, **throw** を使用できます。try {...} の中で throw が行われると try {...} 中の処理は中断され catch {...} が実行され、最後に finally {...} が実行されます。

**try {**

**throw** "Validation Error"

**}**

**catch {**

Write-Host ("ERROR: " + $\_)

**}**

**finally {**

Write-Host "FINALLY"

**}**

##### trap文

**trap** は、シンタックスエラーなどのエラーが発生した際に呼ばれるロジックを定義します。

**trap** {

Write-Host "Trap!!"

break

**}**

$a = 1 / 0 # 0割エラーでtrapルーチンが呼び出される

##### exit文

**exit** はスクリプトを終了します。終了コードには Int32 の値を指定できます。省略すると 0 となります。終了コードは $LastExitCode で参照できます。

if ($error\_code) {

**exit** $error\_code

}

#### 関数

##### 関数定義(function)

**function** は関数を定義します。

**function** Write-Hello {

Write-Output "Hello world!"

}

Write-Hello

##### 引数(param)

関数はパラメータを受け取ることもできます。

function Write-Hello **($Name, $Color)** {

Write-Host "Hello **$Name**!" -ForegroundColor **$Color**

}

Write-Hello "Yamada" "red"

関数を呼び出す際は **括弧無し**、**カンマ無し**で呼び出します。下記の様に呼び出すと ("Yamada", "red") という一つの配列変数を $Name に渡してしまうことになります。

Write-Hello("Yamada", "red") # ERROR

引数で演算等を行いたい場合は、その引数についてのみ括弧でくくります。

Write-Hello ($FirstName + " " + $LastName) "red"

引数を **param** で指定することもできます。こちらの形式が推奨されています。

function Write-Hello {

**param (**

$Name,

$Color

**)**

Write-Host "Hello $Name!" -ForegroundColor $Color

}

Write-Hello "Yamada" "red"

##### 引数の型

下記の様にして引数の型を明示することができます。自動変換できない型の引数を指定した場合はエラーとなります。

function Add(**[int]**$x, **[int]**$y) { $x + $y }

function Sub { param (**[int]**$x, **[int]**$y); $x - $y }

Add 5 3 # 8

Add 5 "ABC" # ERROR

Sub 5 3 # 2

Sub 5 "ABC" # ERROR

##### 引数のデフォルト値

引数のデフォルト値を下記の様に指定することができます。

function Write-Hello($Name, $Color **= "red"**) { ... }

function Write-Hello {

param ($Name, $Color **= "red"**)

...

}

##### 必須引数(Mandatory)

引数は関数内で参照しなければ省略可能です。引数を必須とする場合は下記の様に宣言します。

function Write-Hello {

param (**[Parameter(Mandatory)]**$Name, $Color)

...

}

##### バリデートセット(ValidateSet)

**ValidateSet** により引数が取りうる値を制限することができます。下記の例では -Color 引数を "Red", "Green", "Blue" のいずれかに限定しています。

function Write-Hello {

param (

[Parameter(Mandatory)]$Name,

**[ValidateSet("Red", "Green", "Blue")]**$Color

)

...

##### バリデートスクリプト(ValidateScript)

**ValidateScript** により引数をチェックするスクリプトを記述することができます。下記の例では引数 $num の値を偶数に限定しています。

function Test {

param (

**[ValidateScript({**$\_ % 2 -eq 0**})]**$Num

)

...

}

##### 可変引数(ValueFromRemainingArguments)

下記の様にして可変引数を受け取ることができます。

function Write-Format {

param (

[Parameter(Mandatory)]$Format, # 最初の引数を得る

**[Parameter(ValueFromRemainingArguments)]**$ArgList # 2個目以降の引数リストを得る

)

...

}

もしくは、引数のリストを格納する **$args** を参照することもできます。

function Write-Format {

foreach ($a in **$args**) {

Write-Host "==== $a"

}

}

##### 参照渡し([ref])

**[ref]** を用いることで変数を参照渡しすることができます。参照渡しした変数は、関数内部でその値(Value)を書き換えることができます。

function Test(**[ref]**$a) {

$a**.Value** = 456

}

$a = 123

Test (**[ref]**$a)

Write-Host $a # 456

##### 戻り値(return)

**return** は関数の戻り値を指定します。

function Say-Hello($Name) {

**return** "Hello $Name!"

}

Write-Host (Say-Hello("Yamada")) # Hello Yamada!

return を省略すると最後の式の値が返却値となります。

function Say-Hello($Name) {

**"Hello $Name!"**

}

Write-Host (Say-Hello("Yamada")) # Hello Yamada!

下記の様に複数の値を返却することもできます。

function ToLowerAndUpper($Str) {

return $str.ToLower(), $str.ToUpper()

}

**$lower, $upper** = ToLowerAndUpper("Yamada")

Write-Host $lower # yamada

Write-Host $upper # YAMADA

##### 引数の名前指定

呼び出し側は下記の様に -引数名 を指定して呼び出すことができます。

Write-Hello **-Color** "red" **-Name** "Yamada"

#### フィルタリング

##### 条件抽出(Where/Where-Object)

**Where** は条件抽出を行います。下記の例では作成日が 2023/11/20 00:00:00 以降のファイルを表示します。

Get-ChildItem | **Where {** $\_.LastWriteTime -gt "2023/11/20 00:00:00" **}**

**Where-Object** も同様の動きをします。

Get-ChildItem | **Where-Object {** $\_.LastWriteTime -gt "2023/11/20 00:00:00" **}**

##### フィルタ(filter)

関数と似ていますが、**filter** はフィルタを定義します。下記の例では引数で指定した拡張子を持つファイルのみをフィルタリングします。

**filter** Select-Extension($ext) {

if ($\_.Extension -eq $ext) {

return $\_

}

}

Get-ChildItem | Select-Extension ".txt"

#### クラス(class)

クラスは下記の様に定義します。

**class** MyClass **{** # クラス定義

**[string]$Name** # メンバ変数

**WriteName() {** # クラスメソッド

Write-Host $this.Name

**}**

}

$obj = **[MyClass]::new()** # インスタンス生成

$obj**.Name** = "Yamada" # メンバ変数アクセス

$obj**.WriteName()** # クラスメソッド呼び出し

##### コンストラクタ

クラスメイト同じ名前を持つメソッドはコンストラクタとしてインスタンス作成時に呼ばれます。7.4.0 時点ではデストラクタはサポートされていないようです。

class MyClass {

[string]$Name

**MyClass(**[string]$Name**) {** # コンストラクタ

$this.Name = $Name

**}**

}

$obj = [MyClass]::new("Yamada")

Write-Host $obj.Name # "Yamada"

##### スタティックメソッド(static)

**static** はスタティックメソッドを定義します。スタティックメソッドはクラスをインスタンス化しなくても [*ClassName*]::*Method*() で呼び出すことができます。

class MyClass {

**static** Hello() { # スタティックメソッド

Write-Host "Hello!"

}

}

**[MyClass]::**Hello()

##### 隠しパラメータ(hidden)

メンバ変数に **hidden** をつけると、インスタンスを表示した際に表示されなくなります。

class MyClass {

$param1

**hidden** $param2

}

$obj = [MyClass]::new()

$obj.param1 = "ABC"

$obj.param2 = "XYZ"

$obj # param1 は表示されるが、param2 は表示されない

##### クラスの継承

下記の様にして親クラスを子クラスが継承することができます。

class MyParentClass { hello() { Write-Host "Hello" } }

class MyChildClass **: MyParentClass** { }

$obj = [MyChildClass]::new()

$obj.hello()

#### スコープ

関数の中で定義された変数はローカルスコープの変数とみなされ、関数外の変数とは別のものとなります。

function func() {

**$a = 456** # func の中だけで有効なローカル変数

}

$a = 123

func

Write-Host $a # 123

他の言語では if文や、for文でもローカルスコープとなりますが、PowerShell の場合は if, for, while などではローカルスコープを生成しません。

$a = 123

if ($true) {

**[int]$a = 456** # if文に閉じたスコープは生成しない

}

Write-Host $a # 456 (他の言語では123になるものが多い)

**global:** スコープ修飾子は変数がグローバルスコープのグローバル変数であることを示します。

$**global**:a = 123

スコープ修飾子には下記などがあります。

$**global:**a = 123 # すべてで利用可能なグローバルスコープ

$**script:**a = 123 # スクリプトファイル内で有効なスコープ

$**private:**a = 123 # モジュール内で有効なスコープ

$**local:**a = 123 # 関数やスクリプトブロック内で有効なローカルスコープ

$**using:**a = 123 # Start-Job や Invoke-Command で渡される変数スコープ

また、使用する PSDrive プロバイダによって下記などのスコープ修飾子が利用可能となります。

**Alias:** # 現スコープで利用可能なエイリアス

**Env:** # 現スコープで利用可能な環境変数

**Function:** # 現スコープで利用可能な関数

**Variable:** # 現スコープで利用可能な変数

**HKLM:** # レジストリ(HKEY\_LOCAL\_MACHINE)

**HKCU:** # レジストリ(HEKY\_LOCAL\_USER)

PS C:\> Write-Host **$Env:PATH** # 環境変数PATHの内容を表示する

PS C:\> Set-Location **Env:** # Env:スコープに移動する

PS Env:\> Get-ChildItem # 環境変数の一覧が表示される

PS Env:\> Set-Location **C:** # Cドライブスコープに戻る

PS C:\>

#### パイプとリダイレクト

##### パイプ(|)

コマンドの実行結果をパイプ(|)で次のコマンドに渡すことができます。

Write-Output "C:\" **|** Get-ChildItem

Get-ChildItem -Path \*.txt **|** Get-Content

##### プロセス(process)

**process** を用いてパイプから受け取った各配列に対して処理を行うことができます。

function Write-Double-Value {

**process {** Write-Host ($\_ \* 2) **}**

}

1, 2, 3, 4, 5 | Write-Double-Value # 2, 4, 6, 8, 10

##### 開始処理・終了処理(begin, end)

**begin**, **end** を用いると最初と最後のみに実行する処理を記述することができます。

function Write-Double-Value {

**begin {** Write-Host "----BEGIN----" **}**

process { Write-Host ($\_ \* 2) }

**end {** Write-Host "----END----" **}**

}

1, 2, 3, 4, 5 | Write-Double-Value

##### リダイレクト(>)

コマンドの実行結果をファイルにリダイレクトして書き込むことができます。

Write-Output "Hello" **>** file.txt # 成功ストリームを file.txt に書き込む

Write-Output "Hello" **>>** file.txt # 成功ストリームを file.txt に追記する

Write-Error "ERROR" **2>** file.txt # エラーストリームを file.txt に書き込む

Write-Error "ERROR" **2>&1 >** file.txt # エラーストリームを成功ストリームに振り向けて file.txt に書き込む

ここで 1 や 2 などの数字は下記を意味します。

1 : 成功ストリーム(Linuxで言うところの標準出力) Write-Output

2 : エラーストリーム(Linuxで言うところの標準エラー出力) Write-Error

3 : 警告ストリーム Write-Waring

4 : 詳細ストリーム Write-Verbose

5 : デバッグストリーム Write-Debug

6 : 情報ストリーム Write-Information Write-Host

\* : すべてのストリーム

#### コマンドレット

##### 主なコマンドレット

ヘルプ：

Get-Help ヘルプを表示する

Get-Command コマンドの一覧を取得する

Show-Command コマンドの引数入力ダイアログを表示する

出力：

Write-Host ホスト(コンソール)に文字列情報を出力する

Clear-Host ホスト(コンソール)をクリアする

Write-Output 成功ストリームにオブジェクト情報を出力する

Write-Error エラーストリームにオブジェクト情報を出力する

Write-Warning 警告ストリームにオブジェクト情報を出力する

Write-Verbose 詳細ストリームにオブジェクト情報を出力する

Write-Debug デバッグストリームにオブジェクト情報を出力する

Write-Information 情報ストリームにオブジェクト情報を出力する

Write-Progress 進行状況バーに進行状況を表示する

Out-GridView 別ウィンドウにグリッドビューとして表示する

Out-Null 出力を捨てる

ロケーション(カレントディレクトリ)：

Get-Location カレントディレクトリ情報を取得する

Set-Location カレントディレクトリを変更する

Push-Location カレントディレクトリを移動し、古いディレクトリをプッシュする

Pop-Location プッシュしたディレクトリに戻る

アイテム(ファイル、ディレクトリ、エイリアス、リンク、環境変数等)：

Get-Item アイテム情報を取得する

Get-ChildItem 子アイテムを取得する

New-Item 新しいアイテムを作成する

Set-Item アイテムの値を設定する

Copy-Item アイテムをコピーする

Move-Item アイテムを移動する

Rename-Item アイテムを名前変更する

Remove-Item アイテムを削除する

Clear-Item アイテムの値をクリアする(アイテム自体は削除しない)

Invoke-Item アイテムを実行する

アイテムプロパティ(ファイル作成日やサイズなど)：

Get-ItemProperty アイテムプロパティ情報を得る

Get-ItemPropertyValue アイテムプロパティの値を得る

Set-ItemProperty アイテムプロパティを設定する

New-ItemProperty アイテムプロパティを作成する

Copy-ItemProperty アイテムプロパティをコピーする

Move-ItemProperty アイテムプロパティを移動する

Clear-ItemProperty アイテムプロパティをクリアする

Rename-ItemProperty アイテムプロパティを名前変更する

Remove-ItemProperty アイテムプロパティを削除する

コンテント(ファイルの中身など)：

Get-Content コンテントを得る

Set-Content コンテントを設定する

Add-Content コンテントを追加する

Clear-Content コンテントをクリアする

変数(環境変数)：

Get-Variable 変数情報を得る

Set-Variable 変数を設定する

New-Variable 変数を作成する

Clear-Variable 変数をクリアする

Remove-Variable 変数を削除する

エイリアス：

Get-Alias エイリアス情報を取得する

Set-Alias エイリアスを設定する

New-Alias エイリアスを作成する

Remove-Alias エイリアスを削除する

Export-Alias エイリアス情報をファイルにエクスポートする

Import-Alias エイリアス情報をファイルからインポートする

パス：

Join-Path パス名を連結する

Split-Path パス名を分離する

Convert-Path パス名を変換する(相対パスを絶対パスになど)

Resolve-Path パス名を解決する(ワイルドカード展開など)

Test-Path パス名をテストする(存在確認など)

オブジェクト：

New-Object オブジェクトを作成する

Compare-Object オブジェクトを比較する

Sort-Object オブジェクトをソートする

Select-Object オブジェクトを選択する

ForEach-Object オブジェクトの各項目に対して処理を実行する

Where-Object プロパティ値によってオブジェクトを選択する

Group-Object プロパティ値によってオブジェクトをグルーピングする

Measure-Object オブジェクトの行数・単語数・文字数を計算する

Tee-Object オブジェクトをファイルや変数に格納しながら次のパイプラインに渡す

ヒストリ：

Get-History ヒストリを取得する

Clear-History ヒストリをクリアする

Add-History ヒストリを追加する

Invoke-History ヒストリを実行する

プロセス

Get-Process プロセス情報を得る

Start-Process プロセスを開始する

Stop-Process プロセスを終了する

Wait-Process プロセスの停止を待つ

Debug-Process プロセスをデバッグする

ジョブ(バックグランドプロセス)：

Get-Job ジョブ情報を得る

Start-Job ジョブを開始する

Stop-Job ジョブを終了する

Debug-Job ジョブをデバッグする

Receive-Job ジョブ結果を得る

Wait-Job ジョブの終了を待つ

Remove-Job ジョブを削除する

日時・タイムゾーン：

Get-Date 日時を得る

Set-Date 日時を設定する

Get-TimeZone タイムゾーン情報を得る

Set-TimeZone タイムゾーンを設定する

文字列操作：

Select-String 文字列を検索する

Out-String オブジェクトを文字列に変換する

Join-String オブジェクトを連結して文字列化する

ConvertFrom-StringData キー=バリュー形式の文字列をハッシュテーブルに変換する

フォーマット：

Format-List リスト形式にフォーマットする

Format-Table テーブル形式にフォーマットする

Format-Wide ワイド形式にフォーマットする

Format-Custom 独自形式にフォーマットする

Format-Hex 16進ダンプする

Export-Csv CSVファイルにエクスポートする

Import-Csv CSVファイルからインポートする

ConvertTo-Csv CSV形式にコンバートする

ConvertFrom-Csv CSV形式からコンバートする

ConvertTo-Json JSON形式にコンバートする

ConvertFromo-Json JSON形式からコンバートする

Test-Json JSON形式をテストする

Export-Clixml XMLファイルにエクスポートする

Import-Clixml XMLファイルからインポートする

ConvertTo-Xml XMLにコンバートする

Select-Xml XMLを検索する

ConvertTo-Html HTMLにコンバートする

ConvertFrom-Markdown Markdownからコンバートする

Show-Markdown Markdownを表示する

モジュール：

Get-Module モジュールの一覧を得る

New-Module モジュールを作成する

Import-Module モジュールをインポートする

Remove-Module モジュールを削除する

Find-Module モジュールを検索する

Install-Module モジュールをインストールする

Update-Module モジュールをアップデートする

Uninstall-Module モジュールをアンインストールする

Save-Module モジュールを保存する

Publish-Module モジュールを公開する

クリップボード：

Get-Clipboard クリップボード情報を得る

Set-Clipboard クリップボードを設定する

Webアクセス：

Invoke-WebRequest Webリクエストを発行する

Invode-RestMethod RESTリクエストを発行する

その他

Start-Sleep スリープする

Get-Unique 重複情報をユニーク化する

Get-ComputerInfo コンピュータ情報を得る

Get-Random ランダム値を得る

Get-Error 最後に発生したエラー情報を得る

Invoke-Expression 指定した文字列をコマンドや式として実行する

Compress-Archive ZIP圧縮する

Expand-Archive ZIP解凍する

注：一部コマンドレットではなく関数として実装されているのもあります。

##### Linuxコマンドに対応するコマンドレット

alias Get-Alias, Set-Alias, New-Alias, Remove-Alias

cat Get-Content

cd Set-Location

cp Copy-Item

curl Invoke-WebRequest, Invoke-RestMethod

crear Clear-Host

date Get-Date, Set-Date

df Get-PSDrive

diff Compare-Object

dir Get-ChildItem

echo Write-Host, Write-Output

env Get-Variable, Set-Variable, ...

find Get-ChildItem -Recurse

for ForEach-Object

grep Select-String

head Get-Content -Head *n*

history Get-History, Invoke-History, ...

jobs Get-Job, Stop-Job, ...

kill Stop-Process

ls Get-ChildItem

man Get-Help

mkdir New-Item *dir* -ItemType Directory

mv Move-Item, Rename-Item

push Push-Location

pop Pop-Location

ps Get-Process

pwd Get-Location

rm Remove-Item

rmdir Remove-Item

set Get-Variable, Set-Variable, ...

sleep Start-Sleep

sort Sort-Object

tail Get-Content -Tail *n*

tee Tee-Object

uniq Get-Unique

wc Measure-Object

#### その他

##### 正規表現

**Matches()** を用いて正規表現にマッチする複数の文字列を得ることができます。

$a = **[regex]::Matches(**"23:59:59", "\d{2}"**)**

Write-Host ("{0}時{1}分{2}秒" -f $a[0], $a[1], $a[2]) # 23時59分59秒

**Replace()** を用いて正規表現にマッチする部分文字列を置換します。

$a = **[regex]::Replace(**"23:59:59", "\d{2}", "XX"**)**

Write-Host $a # XX:XX:XX

##### フォーマット

下記の様にしてフォーマットを指定することができます。

(1234567890).ToString(**"#,0"**) # 1,234,567,890

(123.4).ToString(**"0.000"**) # 123.400

(255).ToString(**"x4"**) # 00ff

(255).ToString(**"X4"**) # 00FF

(Get-Date).ToString(**"yyyy/MM/dd HH:mm:ss"**) # 2023/11/23 22:29:42

**-f** でフォーマットを指定します。{*n*} の *n* は -f の後ろに指定した引数の順序(0始まり)を示します。

"**{0}**時**{1}**分**{2}**秒" **-f** $hour, $min, $sec # 23:59:59

{*n*, *s*} の *s* は桁数・文字数を指定します。正数を指定すると左詰め、負数を指定すると右詰になります。

"[**{0,5}**][**{1,-5}**]" -f "ABC", "XYZ" # [ ABC][XYZ ]

{*n*:*f*, *s*} の *f* にフォーマットを指定することもできます。

"{0:**#,#**}" -f 12345 # 12,345 - カンマ区切り数値

"{0:**0000**}" -f 255 # 0244 - 0埋め4桁整数

"{0:**0.00**}" -f 255 # 255.00 - 小数第2位まで表示

"{0:**F2**}" -f 255 # 255.00 - 小数第2位まで表示

"{0:**X4**}" -f 255 # 00FF - 4桁16進数(大文字)

"{0:**x4**}" -f 255 # 00ff - 4桁16進数(小文字)

[String]::Format() を使用することもできます。

**[String]::Format(**"{0}時{1}分{2}秒", $hour, $min, $sec**)** # 23:59:59

##### ファイルを読み込む

ファイルを逐次読み込むには StreamReader() や RealLine() を使用します。

$f = **New-Object System.IO.StreamReader(**"C:\Temp\xx.txt", [System.Text.Encoding]::GetEncoding("utf-8")**)**

while (($line = $f**.ReadLine()**) -ne $null) {

Write-Host "> $line"

}

$f**.Close()**

##### コンソールから読み込む

コンソールから文字列を読み込むには **PSConsoleHostReadLine** を用います。

Write-Host "Please input your name:"

$name = **PSConsoleHostReadLine**

Write-Host "Hello $name."

##### 一時停止(Pause)

**Pause** は Enter キーを押すまで一時停止します。

**Pause**

##### ストリクトモード

スクリプトの先頭で下記を実行すると、未定義の変数を参照した場合などにエラーとなり、変数名誤りなどによるバグを減らすことができます。

**Set-PSDebug -Strict**

##### プロンプト(prompt)

prompt 関数を定義するとプロンプトを変更することができます。

function **prompt {** "PS> " **}**

デフォルトでは下記の様に定義されています。

function **prompt {** "PS $($executionContext.SessionState.Path.CurrentLocation)$('>' \* ($nestedPromptLevel + 1)) " **}**

##### プロファイル($profile)

プロファイルのパス名は **$profile** で確認できます。プロファイルにコマンドや関数をを記述しておくと PowerShell 起動時に読み込まれます。

PS C:\> **$profile**

C:\Users\yamada\Documents\PowerShell\Microsoft.PowerShell\_profile.ps1

##### コマンド置換($(...))

$(...) の中にコマンドを記述するとコマンドの実行結果を文字列やオブジェクトとして扱うことができます。

Write-Host "DEBUG: " **$(Get-Date)**

foreach ($n in **$(Get-ChildItem)**) { $i }

宿題：

1. C:\work\shukudaiに下記ファイルがあります。

証跡\_【画面ID】\_画面ごとの順番.拡張子

証跡\_G0001\_01.txt

証跡\_G0001\_02.txt

証跡\_G0002\_01.txt

証跡\_G0002\_02.txt

証跡\_G9000\_01.txt

PowerShellで作業フォルダーの直下に画面IDごとサブフォルダーを作成し、ファイルをそれぞれのサブフォルダーへ移動してください。