

# PowerShell → typowe komendy

## STRUKTURA KOMEND

`verb + noun` (czasownik + rzeczownik)

`-verb` → `get-command -verb get` → lista dostępnych komend z `get` jako czasownik

`-noun` → `get-command -noun host` → komendy z `host` jako rzeczownik

Jak napisać więcej linijek w terminalu? `@ " "@`

```
PS C:\Users\mikal\desktop\pliki z zajec\semestr 2\programowanie skryptowe\  
Component;GenerateReport  
Computername;True  
Manufacturer;True  
sSerialNumber;True  
CpuName;False  
RAM;True  
"@ >> input_file.csv
```

Opcja 2: `'`

`'` → na końcu jednej lini, nie trzeba zamykać w skryptach

```
New-Item -Path "ścieżka_do_pliku\nazwa_pliku.txt" '  
-ItemType "File"
```

## POMOC

`get-command` → zwraca listę dostępnych komend

`clear-host` → wyczyszczenie okna

`tab` → dokończenie komendy

`get-history` → historia komend

`strzałka w górę` → poprzednie komendy po kolei

---

## PLIKI/KATALOGI

\*aktualna ścieżka

`Get-Location` ( `pwd` też działa, jest ustawione jako alias)

- WYPISANIE ZAWARTOŚCI PLIKU

`Get-Content input_file.csv`

- TWORZENIE PLIKU/KATALOGU

`New-Item -Path <ścieżka> -ItemType <typ_elementu>`

Plik: `New-Item -Path "ścieżka_do_pliku\nazwa_pliku.txt" -ItemType "File"`

Katalog: `New-Item -Path "C:\Users\mikoł\NowyFolder" -ItemType "Directory"`

- ZMIANA NAZWY PLIKU/KATALOGU

`Rename-Item -Path <ścieżka> -NewName <nowa_nazwa>`

Przykład: `Rename-Item -Path .\test.ps1 -NewName testt.ps1`

- OBLICZANIE HASHU PLIKU (Domyślnie: SHA256)

`Get-FileHash -Path <ścieżka> [-Algorithm <algorytm>]`

Przykład: `Get-FileHash -path C:\Users\mikoł\testt.ps1`

- WYLISTOWANIE ZAWARTOŚCI KATALOGU (takie `ls` z linuxa)

*\*ls jest domyślnie aliasem, więc też działa*

`Get-ChildItem`

`-Path` : określa folder, który listujemy (bez tego, listuje aktualny z `pwd` )

`Get-ChildItem -Path "C:\Users\Użytkownik\Documents"`

`-Recurse` : rekurencyjne wyświetlanie (wraz z podkatalogami)

`Get-ChildItem -Recurse`

`-File / -Directory` → wyświetla tylko pliki/katalogi

`-Name` → wyświetla same nazwy

`-Filter "*.pdf"` → pliki .pdf

- PRZEKIEROWANIE DO PLIKU

`text/komenda | Out-File -FilePath <ścieżka_do_pliku>` → wymienia zawartość pliku

`-Append` → dopisuje do pliku

Linuxowe też działa:

`text/komenda > <ścieżka_do_pliku>` → nadpisuje

`text/komenda >> <ścieżka_do_pliku>` → dodaje

---

## PROCESY/PROGRAMY

- Lista procesów:

`Get-Process`

- Uruchomienie procesu/programu

`Start-Process -FilePath <ścieżka_do_programu> [-ArgumentList <argumenty>]`

`Start-Process notepad.exe`

`Start-Process -path "C:\ProgramData\Microsoft\Windows\Start Menu\Programs\Epic Games Launcher.lnk"`

---

## INFORMACJE O KOMPIE /UŻYTKOWNIKACH/ SYSTEMIE

`get-host` → info o PowerShellu

`Get-Date` → piątek, 4 kwietnia 2025 21:10:58

`Get-Date -format "yyyymmdd-HH:mm:ss"`

▼ wyjaśnienie

- **yyyy** – czterocyfrowy rok (np. 2025),
- **MM** – dwucyfrowy miesiąc (np. 04),
- **dd** – dwucyfrowy dzień (np. 11),
- **HH** – godzina (00-23),
- **mm** – minuta,
- **ss** – sekunda,
- **fff** – milisekundy.

`Get-ComputerInfo` → ogrom informacji o soft/hardware, domyślnie jest wypisywana tylko część

`Get-CimInstance -ClassName <NazwaKlasy>` → znowu dużo informacji, wymaga podania klasy, nie ładuje się tak długo jak `Get-ComputerInfo`

#### ▼ klasy

`Get-CimClass | Select-Object -ExpandProperty CimClassName` → pełna lista

## System operacyjny i komputer

Klasa CIM	Opis
<code>Win32_OperatingSystem</code>	Informacje o systemie operacyjnym (nazwa, wersja, architektura, uptime)
<code>Win32_ComputerSystem</code>	Dane komputera (model, producent, ilość pamięci RAM)
<code>Win32_BIOS</code>	Informacje o BIOS-ie/UEFI
<code>Win32_StartupCommand</code>	Programy uruchamiane przy starcie systemu
<code>Win32_TimeZone</code>	Informacje o strefie czasowej

## Sprzęt

Klasa CIM	Opis
<code>Win32_Processor</code>	Szczegóły procesora (nazwa, taktowanie, liczba rdzeni)
<code>Win32_PhysicalMemory</code>	Informacje o kościach RAM (pojemność, producent, częstotliwość)
<code>Win32_VideoController</code>	Dane o karcie graficznej
<code>Win32_SoundDevice</code>	Informacje o karcie dźwiękowej
<code>Win32_Keyboard</code>	Szczegóły klawiatury

Win32_PointingDevice	Mysz i inne urządzenia wskazujące
Win32_Battery	Informacje o baterii (w laptopach)

## Dyski i partycje

Klasa CIM	Opis
Win32_DiskDrive	Lista fizycznych dysków (HDD, SSD)
Win32_LogicalDisk	Informacje o partycjach (literki dysków, wolna przestrzeń)
Win32_DiskPartition	Szczegóły partycji
Win32_Volume	Informacje o woluminach, systemie plików

## Sieć

Klasa CIM	Opis
Win32_NetworkAdapter	Lista kart sieciowych
Win32_NetworkAdapterConfiguration	Konfiguracja sieci (adresy IP, MAC, DNS)
Win32_IP4RouteTable	Tabela routingu IPv4
Win32_TCIPPrinterPort	Porty drukarek sieciowych
Win32_PingStatus	Możliwość pingowania hostów

## Oprogramowanie i procesy

Klasa CIM	Opis
Win32_Process	Lista aktualnie działających procesów
Win32_Service	Lista usług systemowych
Win32_Product	Lista zainstalowanych programów
Win32_StartupCommand	Programy uruchamiane przy starcie systemu

## Użytkownicy i zabezpieczenia

Klasa CIM	Opis
Win32_UserAccount	Lista kont użytkowników
Win32_Group	Grupy użytkowników
Win32_LogonSession	Aktywne sesje logowania

## Przykłady wydobywania jakiś danych:

### ▼ Nazwa komputera (hosta)

`hostname` → terminal

`$(hostname)` → skrypty

### ▼ Nazwa systemu operacyjnego (+wersja)

`$os = Get-ComputerInfo | Select-Object OsName, OsVersion`

### ▼ Nazwa aktualnego użytkownika

`$name = $env:USERNAME`

### ▼ Adres IP(v4)

`$ip=Get-NetIPAddress -AddressFamily IPv4 | Where-Object {$_.InterfaceAlias -eq "Ethernet"} | Select-Object IPAddress`

### ▼ Nazwa producenta urządzenia, model urządzenia

```
$info1 = Get-CimInstance -ClassName Win32_ComputerSystem
$Manufacturer = $info1.Manufacturer
$Model = $info1.Model
```

### ▼ Nazwa CPU

`$CpuName = Get-WmiObject -Class Win32_Processor | Select-Object Name`

### ▼ RAM (łącznie, GB)

`[string]$RAM = ((Get-ComputerInfo).CsTotalPhysicalMemory /1GB).ToString() + " GB"`

## Zmienne Środowiskowe

→ informacje o aktualnym środowisku, danej sesji (np. PowerShella), są to zmienne globalne

`Get-ChildItem Env:` → wypisuje zmienne środowiskowe

`$Env:USER = "Mikołaj"` → tworzy zmienną USER (konieczny caps lock!) o wartości *Mikołaj*

`$Env:USER` → wypisuje wartość zmiennej USER

## SORTOWANIE

`Sort-Object -Property <właściwość> [-Descending]`

Malejąco: `coś | Sort-Object -descending`

Wg. właściwości ( `property` )(co to? niżej jest o obiektach)

`Get-ChildItem | Sort-Object -property Name` → `GetChildItem` ma właściwość (kolumnę) "Name", sortujemy wg. niej

`Get-Process | Sort-Object -Property WorkingSet` → `Get-Process` ma właściwość (kolumnę) "WorkingSet"

Bez duplikatów: `-Unique`

Z rozróżnianiem wielkości liter: `-CaseSensitive`

---

## WYCINANIE

- Wycinanie właściwości (kolumn) `Select-Object`

`Get-Process | Select-Object Name, Id` → tylko kolumny *Name* i *Id*

`-Skip 5` → pomija 5 pierwszych wyników

`-First 3` → Pierwsze 3 wyniki

`-Last 10` → Ostatnie 10 linijek

`Get-Process | select-object *` → wyświetla `get-process` z WSZYSTKIMI właściwościami, nie tylko domyślnymi

`Get-ChildItem | Select-Object Name, @{Name="FileSizeKB"; Expression={$_.Length / 1KB}}`

`@{Name="FileSizeKB"; Expression={$_.Length / 1KB}}` → tworzymy własną kolumnę

`Name="FileSizeKB"` → nazwa kolumny → *FileSizeKB*

`Expression={$_.Length / 1KB}` → wyświetla rozmiar pliku (ale w KB, zamiast bajtach)

`$_` → odwołania do aktualnej linijki procesu (tu: danego pliku)

---

## NASŁUCHIWANIE FOLDERU

(i automatyczne przenoszenie rzeczy do innego folderu)

Opcja 1 (sprawdza folder cały czas, proste, ale dużo zasobów):

```
$zrodlo = "sciezka/zrodlo"
```

```
$cel = "sciezka/cel"
```

```
Write-Host "Trwa prosty monitoring folderu '$zrodlo'. Naciśnij Ctrl+C, aby zak
```

```
while ($true){
```

```
$pliki = Get-ChildItem $zrodlo #-Filter *.txt -File jak tylko .txt
```

```
foreach($plik in $pliki){
```

```
Move-Item -Path $plik.FullName -Destination $cel #bez pelnej sciezki (.FullNa
```

```
Write-Host "Przeniesiono $($plik.Name)"
```

```
}
```

```
}
```

Opcja2 (bardziej skomplikowana składnia, działa dopiero po dodaniu pliku (natychmiast))

```
# Tworzymy obiekt FileSystemWatcher
```

```
$watcher = New-Object System.IO.FileSystemWatcher
```

```
# Określamy folder, który chcemy monitorować
```

```
$watcher.Path = "C:\Users\mikoł\Desktop\pliki z zajęć\semestr 2\programowa
```

```
$watcher.Filter = "*.txt" #ewentualne filtry
```

```
# Włączamy monitorowanie
```

```
$watcher.EnableRaisingEvents = $true
```

```
#akcja, gdy plik zostanie znaleziony
```

```
Register-ObjectEvent $watcher Created -Action{
```

```
# Created → akcja, inne to np. Changed/Deleted -Action { ... } → pole akcji
```

```
Start-Sleep -Seconds 2
```

```
#delay, np. na ustalenie nazwy w GUI, przetworzenie przez komputer
```

```
$plik = $Event.SourceEventArgs.FullPath
```



```

#uzyskanie pelnej sciezki argumentow (plikow, ktore watcher zobaczyl)
$cel = "C:\Users\mikoł\Desktop\pliki z zajęć\semestr 2\programowanie skry
Move-Item -Path $plik -Destination $cel
Write-Host "Przeniesiono plik: $plik"
}

# Skrypt działa, dopóki nie zostanie przerwany
Write-Host "Monitorowanie folderu... Naciśnij Ctrl+C, aby zakończyć."
while ($true) {
    Start-Sleep -Seconds 1 #watcher aktualizuje się co sekundę
}

```

## OBIEKTY, WŁAŚCIWOŚCI, METODY

### OBIEKT (OBJECT)

Podstawowa jednostka danych, w zasadzie wszystko w PowerShellu to obiekt; liczby, ciągi znaków, stringi, procesy....

**ELEMENTY/CZŁONKOWIE (MEMBERS)** → informacje, możliwości, działania związane z obiektem

Aby wypisać ich listę, wrzucamy *obiekt* do `get-member` :

`"nazwa_obiektu" | get-member`

Przykład, w którym PowerShell wykrywa, że obiekt to string, i wyświetla listę `members` dla stringów:

```

PS C:\> "Codecademy" | Get-Member

TypeName: System.String

Name      MemberType      Definition
-----
Clone     Method          System.Object Clone(), System.Object ICloneable.Clone()
CompareTo Method          int CompareTo(System.Object value), int CompareTo(Stri..
Contains  Method          bool Contains(string value)
CopyTo    Method          void CopyTo(int sourceIndex, char[] destination, int d..
EndsWith  Method          bool EndsWith(string value), bool EndsWith(string valu..
Equals    Method          bool Equals(System.Object obj), bool Equals(string valu..
GetEnumerator Method          System.CharEnumerator GetEnumerator(), System.Collecti..

```

- **WŁAŚCIWOŚCI (PROPERTIES)** → cechy obiektu, informacje o nim

Przykład ze screena wyżej, jedyna właściwość stringów:

Length	Property	int Length {get;}

"nazwa\_obiektu" | get-member -Member-Type Property → zwraca same właściwości

\* -type zadziała, ale nie jest poprawne

Odwołanie się do właściwości:

"nazwa\_obiektu".length → 13

- **METODY (METHODS)** → operacja, którą można wykonać na obiekcie

Lista metod → obiekt | get-member -Member-Type Method

Przykładowe metody:

.Get-Type :

\$wiek.GetType().name → zwraca typ obiektu (tutaj: Int32 )

\*bez .name jest więcej info, .name to nazwa właściwości w wyniku GetType()

.contains → "jeżozwierz".contains("jeż") → True

.replace :

\$ja = "kulfon"

\$ja.replace("kul", "cool") → coolfon

Write-host \$ja → kulfon (replace nie zmienia zmiennej, chyba, że: \$ja = komenda )

.CompareTo()

\$liczba=5 → \$liczba.compareto(10) → -1

-1 → \$liczba jest mniejsze

0 → \$liczba jest równe

1 → \$liczba jest większe

```
.ToString() → $string = $liczba.ToString()
```

## CREATING OBJECT

```
$dog = New-Object -TypeName PSCustomObject
```

`-TypeName` → typ obiektu, tak jak np. string

`PSCustomObject` → klasa do tworzenia własnych obiektów

## ADDING PROPERTIES

\*zamiast `property` używamy `note-property`, która pozwala na dynamiczne dodawanie właściwości

```
$dog | Add-Member -MemberType NoteProperty -Name "Name" -Value "Rufus"
```

```
$dog | Add-Member -MemberType NoteProperty -Name "Age" -Value 10
```

`$dog.name` → Rufus

## ADDING METHODS

\*zamiast `Method` używamy `ScriptMethod`

```
$dog | Add-Member -MemberType ScriptMethod -Name "speak" -Value { Write-Host "Woof!" }
```

`$dog.speak()` → Woof!

UWAGA NOWSZY SPOSÓB!

```
$dog = [PSCustomObject]@{  
    Name = "Rufus"  
    Age = 10  
}
```