Introduction to PowerShell ISE

What is PowerShell ISE?

- **ISE (Integrated Scripting Environment)** is a **graphical user interface** for PowerShell.
- It allows users to **write, test, and debug PowerShell scripts** easily.
- It provides features like:
  - Syntax highlighting
  - Auto-completion
  - Multi-line editing
  - Integrated console output window

Why use PowerShell ISE?

- Easier to **create and edit scripts** than using the command line.
- You can **run part of a script** or the **entire script** easily.
- Helps in **debugging** using breakpoints and variable watch.
- Provides **IntelliSense** for commands, parameters, and objects.

How to Open PowerShell ISE

- Search in Start Menu → **Windows PowerShell ISE**
- OR use command:
- `powershell_ise`

PowerShell Data Types

What is a Data Type?

A **data type** defines what kind of value a variable can store, such as:

- Numbers
- Strings
- Boolean (True/False)
- Arrays
- HashTables
- Objects, etc.

Common Data Types in PowerShell

Data Type	Example	Description
[int]	`[int]$a = 10`	Integer (whole number)
[float] / [double]	`[double]$b = 12.5`	Decimal (floating point number)
[string]	`[string]$name = "Neeraj"`	Text or characters
[bool]	`[bool]$x = $true`	Boolean (True or False)

Data Type	Example	Description
[array]	`$arr = @(1,2,3,4)`	Collection of values
[hashtable]	`$hash = @{Name="John"; Age=25}`	Key-value pairs
[datetime]	`[datetime]$d = Get-Date`	Stores date and time

Get Type and Get Value in PowerShell

1. Get the Data Type of a Variable

Use the `.GetType()` method:

```
$a = 100
$a.GetType()
```

Output Example:

```
IsPublic IsSerial Name     BaseType
-------- -------- ----     --------
True     True     Int32    System.ValueType
```

Here, the variable `$a` is of **Int32 (integer)** type.

2. Get Only the Type Name

If you just want the **type name** (not full details):

```
$a = "PowerShell"
$a.GetType().Name
```

Output:

```
String
```

3. Get the Value of a Variable

Simply type the variable name:

```
$name = "Neeraj"
$name
```

Output:

```
Neeraj
```

Or use `Write-Output` or `Write-Host`:

```
Write-Output $name
Write-Host "The value is: $name"
```

Example Summary:

```
# Declare variables
[int]$num = 10
[string]$text = "Hello"
[bool]$isTrue = $true

# Get types
$num.GetType().Name
$text.GetType().Name
$isTrue.GetType().Name

# Get values
Write-Output $num
Write-Output $text
Write-Output $isTrue
```

Operators in PowerShell ISE (With Examples)

Introduction

Operators in PowerShell are used to perform **mathematical, logical, comparison, and assignment** operations on values and variables.
They make it easier to handle data, perform decisions, and build automation scripts.

PowerShell ISE allows you to **test and execute** each operator quickly in its **console pane**.

1. Arithmetic Operators

Used to perform **mathematical operations**.

Operator	Description	Example	Output
+	Addition	5 + 3	8
-	Subtraction	10 - 4	6
*	Multiplication	2 * 6	12
/	Division	15 / 3	5
%	Modulus (remainder)	10 % 3	1
++	Increment	$a++	Increase by 1
--	Decrement	$a--	Decrease by 1

Example Code

```
$a = 10
$b = 3

Write-Host "Addition: " ($a + $b)
Write-Host "Subtraction: " ($a - $b)
Write-Host "Multiplication: " ($a * $b)
Write-Host "Division: " ($a / $b)
Write-Host "Modulus: " ($a % $b)

$a++
```

```
Write-Host "After Increment a = $a"

$b--
Write-Host "After Decrement b = $b"
```

2. Comparison Operators

Used to **compare two values** and return `True` or `False`.
All comparison operators in PowerShell **start with a dash (-)**.

Operator	Description	Example	Output
`-eq`	Equal to	`5 -eq 5`	`True`
`-ne`	Not equal to	`5 -ne 3`	`True`
`-gt`	Greater than	`10 -gt 5`	`True`
`-lt`	Less than	`3 -lt 7`	`True`
`-ge`	Greater than or equal to	`10 -ge 10`	`True`
`-le`	Less than or equal to	`8 -le 9`	`True`

Example Code

```
$a = 15
$b = 20

if ($a -lt $b) {
    Write-Host "$a is less than $b"
}

if ($b -ge 15) {
    Write-Host "$b is greater than or equal to 15"
}

Write-Host "Equality Check: " ($a -eq 15)
Write-Host "Not Equal Check: " ($a -ne $b)
```

3. Logical Operators

Used to **combine multiple conditions**.

Operator	Description	Example	Output
`-and`	True if both are true	`(5 -gt 2) -and (10 -lt 20)`	`True`
`-or`	True if one condition is true	`(5 -gt 10) -or (2 -lt 3)`	`True`
`-not`	True if condition is false	`-not(5 -eq 10)`	`True`
`!`	Same as `-not`	`!(5 -eq 10)`	`True`

Example Code

```
$a = 10
$b = 20
```

```
if (($a -lt $b) -and ($b -gt 5)) {
    Write-Host "Both conditions are True"
}

if (($a -eq 5) -or ($b -eq 20)) {
    Write-Host "At least one condition is True"
}

Write-Host "Not Operator Example: " (-not($a -eq 5))
```

4. Assignment Operators

Used to **assign or update** values in variables.

Operator	Description	Example	Meaning
=	Assign	$a = 5	Assigns 5
+=	Add and assign	$a += 3	$a = $a + 3
-=	Subtract and assign	$a -= 2	$a = $a - 2
*=	Multiply and assign	$a *= 4	$a = $a * 4
/=	Divide and assign	$a /= 2	$a = $a / 2

Example Code

```
$a = 10
Write-Host "Original value of a: $a"

$a += 5
Write-Host "After += : $a"

$a -= 2
Write-Host "After -= : $a"

$a *= 3
Write-Host "After *= : $a"

$a /= 4
Write-Host "After /= : $a"
```

5. String Operators

Used for **comparing or joining strings**.

Operator	Description	Example	Output
+	Concatenate (join)	"Hello " + "World"	Hello World
-eq	Equal strings	"a" -eq "a"	True
-ne	Not equal	"a" -ne "b"	True
-like	Pattern match (*)	"hello" -like "he*"	True
-notlike	Does not match	"hello" -notlike "hi*"	True
-match	Regex match	"PowerShell" -match "Shell"	True

Operator	Description	Example	Output
-notmatch	Not matching regex	"PowerShell" -notmatch "Java"	True

Example Code

```
$str1 = "PowerShell"
$str2 = "Power"

Write-Host "Concatenation: " ($str1 + " Script")
Write-Host "Equal Check: " ($str1 -eq "PowerShell")
Write-Host "Like Check: " ($str1 -like "Power*")
Write-Host "Match Check: " ($str1 -match "Shell")
Write-Host "Not Match Check: " ($str1 -notmatch "Java")
```

6. Array Operators

Used to work with **collections (arrays)**.

Operator	Description	Example	Output
..	Range	1..5	1 2 3 4 5
-contains	Array contains value	@(1,2,3) -contains 2	True
-notcontains	Array does not contain	@(1,2,3) -notcontains 5	True
-in	Value is in array	2 -in @(1,2,3)	True
-notin	Value not in array	5 -notin @(1,2,3)	True

Example Code

```
$arr = 1..5
Write-Host "Array: $arr"

if (3 -in $arr) {
    Write-Host "3 is in the array"
}

if (6 -notin $arr) {
    Write-Host "6 is not in the array"
}
```

7. Type Operators

Used to **check or convert data types**.

Operator	Description	Example	Output
-is	Check type	5 -is [int]	True
-isnot	Not of type	"abc" -isnot [int]	True
[type]	Convert type	[int]"5"	Converts to integer

Example Code

```
$a = "25"
Write-Host "Before Conversion Type: " $a.GetType().Name

$b = [int]$a
Write-Host "After Conversion Type: " $b.GetType().Name

Write-Host "Check Type: " ($b -is [int])
Write-Host "Check Not Type: " ($b -isnot [string])
```

8. Bitwise Operators

Used for **bit-level operations** (mainly on integers).

Operator	Description	Example	Output
-band	Bitwise AND	5 -band 3	1
-bor	Bitwise OR	5 -bor 3	7
-bxor	Bitwise XOR	5 -bxor 3	6
-bnot	Bitwise NOT	-bnot 5	-6
-shl	Shift Left	5 -shl 1	10
-shr	Shift Right	5 -shr 1	2

Example Code

```
$a = 5
$b = 3

Write-Host "Bitwise AND: " ($a -band $b)
Write-Host "Bitwise OR: " ($a -bor $b)
Write-Host "Bitwise XOR: " ($a -bxor $b)
Write-Host "Bitwise NOT of a: " (-bnot $a)
Write-Host "Left Shift: " ($a -shl 1)
Write-Host "Right Shift: " ($a -shr 1)
```

IF–ELSE Statements in PowerShell

An **IF statement** checks a condition.
If the condition is **true**, PowerShell runs the code inside the IF block.

Basic Structure

```
if (condition) {
    # code to run if condition is true
}
```

2. IF–ELSE Statement

Used when you want **two options**:

- If condition is **true**, run IF block.
- If condition is **false**, run ELSE block.

Structure

```
if (condition) {
    # code runs when condition is true
}
else {
    # code runs when condition is false
}
```

Example 1: Check if number is positive

```
$number = 10

if ($number -gt 0) {
    Write-Output "Number is positive."
}
else {
    Write-Output "Number is zero or negative."
}
```

Example 2: Check if a file exists

```
$file = "C:\test.txt"

if (Test-Path $file) {
    Write-Output "File exists."
}
else {
    Write-Output "File does not exist."
}
```

ELSEIF Statement (multiple conditions)

When you have **more than two options**, use `elseif`.

Example 3: Check age category

```
$age = 18

if ($age -lt 13) {
    Write-Output "Child"
}
elseif ($age -lt 20) {
    Write-Output "Teenager"
}
else {
    Write-Output "Adult"
}
```

Important Notes

- Conditions go inside **parentheses ( )**
- Code blocks go inside **curly braces { }**
- Comparison operators:

Operator	Meaning
`-eq`	equal
`-ne`	not equal
`-gt`	greater than
`-lt`	less than
`-ge`	greater or equal
`-le`	less or equal

- PowerShell is **case-insensitive** (`IF`, `If`, `if` all work)

Switch Statement in PowerShell

A **switch statement** is used when you want to test **one value** against **many possible cases**.

It is cleaner than writing many **if–elseif** conditions.

Basic Structure

```
switch (value) {
    case1 { code }
    case2 { code }
    default { code }
}
```

In PowerShell, we do **not** use the word `case`.
We simply write the value:

```
switch ($value) {
    "apple"   { Write-Output "Fruit is apple" }
    "banana"  { Write-Output "Fruit is banana" }
    default   { Write-Output "Fruit not found" }
}
```

Example 1: Day of the Week

```
$day = "Monday"

switch ($day) {
    "Monday"    { "Start of the week" }
    "Friday"    { "Weekend coming soon!" }
    "Sunday"    { "Relax, holiday!" }
    default     { "Normal day" }
}
```

Using Switch with Ranges

You can also use **ranges** using `..` operator:

```
$marks = 85
```

```
switch ($marks) {
    {$_ -ge 90} { "Grade A" }
    {$_ -ge 80} { "Grade B" }
    {$_ -ge 70} { "Grade C" }
    default { "Fail" }
}
```

Here `$_` means *current value being checked*.

Important Notes

- `switch` compares each case **until a match is found**.
- You can use:
  - **Strings**
  - **Numbers**
  - **Expressions** (`{ condition }`)
- `default` is optional, used when no match is found.
- Cleaner and shorter than many `if-elseif`s.

PowerShell Loops

Loops are used to **repeat code** until a condition is met.

PowerShell has **4 main loops**:

1. `for`
2. `foreach`
3. `while`
4. `do-while` / `do-until`

FOR Loop

Used when you know **how many times** you want to repeat.

Structure

```
for (start; condition; step) {
    # code
}
```

Example

```
for ($i = 1; $i -le 5; $i++) {
    Write-Output "Number: $i"
}
```

FOREACH Loop

Used to loop through **each item** in a list or array.

Example

```
$names = @("Ram", "Shyam", "Mohan")

foreach ($n in $names) {
    Write-Output $n
}
```

WHILE Loop

Repeats as long as the **condition is true**.

Example

```
$count = 1

while ($count -le 3) {
    Write-Output "Count: $count"
    $count++
}
```

DO-WHILE Loop

Runs **at least once**, then checks the condition.

Example

```
$x = 1

do {
    Write-Output "Value: $x"
    $x++
} while ($x -le 3)
```

DO-UNTIL Loop

Opposite of while — it runs **until** the condition becomes **true**.

```
$y = 1

do {
    Write-Output $y
    $y++
} until ($y -gt 3)
```

BREAK Statement

`break` stops the loop **immediately**.

Example: stop when i = 3

```
for ($i = 1; $i -le 5; $i++) {
    if ($i -eq 3) {
```

```
        break
    }
    Write-Output $i
}
```

Output:
1
2
(stop here)

CONTINUE Statement

`continue` skips the **current** iteration and moves to the **next** one.

Example: skip number 3

```
for ($i = 1; $i -le 5; $i++) {
    if ($i -eq 3) {
        continue
    }
    Write-Output $i
}
```

Output:
1
2
4
5
(3 is skipped)

Summary (Easy English)

- **for** = loop with counter
- **foreach** = loop through each item
- **while** = loop until condition becomes false
- **do-while** = loop runs at least once
- **break** = stop loop immediately
- **continue** = skip current step and continue next

PowerShell Functions

A **function** is a block of code that you can **reuse**.
You create it once and call it whenever you need.

Basic Function Structure

```
function FunctionName {
    # code to run
}
```

Example

```
function SayHello {
    Write-Output "Hello, PowerShell!"
}

SayHello   # calling the function
```

Function with Parameters

Parameters allow you to send **values** into a function.

```
function Greet {
    param ($name)
    Write-Output "Hello, $name"
}

Greet -name "Neeraj"
```

Multiple Parameters

```
function Add-Numbers {
    param ($a, $b)
    $sum = $a + $b
    return $sum
}

Add-Numbers -a 10 -b 20
```

Function with Return Value

Functions can return values.

```
function GetSquare {
    param ($num)
    return ($num * $num)
}

GetSquare -num 6
```

Advanced Function Example

```
function Check-Age {
    param ($age)

    if ($age -ge 18) {
        "You are an adult"
    }
    else {
        "You are a minor"
    }
}

Check-Age -age 20
```

Notes on Functions

- Functions make scripts **clean** and **reusable**.

- Use `param()` to accept inputs.
- Use `return` to give output (optional — last line also returns value).