Practical work 2 - Windows & PowerShell

Duration	Groups	Evaluation
3 hours	1-2	PDF report on Edunao

Context and instructions

- After exploring the seven seas on your Linux ship, you somehow yearn for a vaccation on land. For this adventure, we'll hop on the Windows train.
- The entirety of the following exercises can be done on a Windows 10/11 install with the default, pre-installed version of Powershell.
- We know LLMs are able to answer the questions. **Please refrain from pasting everything into your favorite LLM and take the time to understand how to use commands and read the associated documentation.** Your future-self will appreciate it.

This lab **can** be done in pairs. To hand in your work:

- Create a **pdf** document (no docx);
- It MUST contain **your names**;
- Upload it on Edunao.

Tips & tricks about Powershell

Today, we'll mainly use Powershell. Like most shells, you can use basic shortcuts to navigate around commands:

- Press `TAB` multiple tabs for **autocompletion**.
- `CTRL` + `L` to **clear the screen**.
- `UP ARROW` and `DOWN ARROW` to **navigate the command history**.
- `CTRL` + `R` to **search** commands in the history.
- `CTRL` + `SHIFT` + `C` to copy, `CTRL` + `SHIFT` + `V` to paste.

To **get help / more information about a command**, use:

- The official Microsoft Windows documentation
- A variation of:

```
help [cmd]
get-help [cmd]
Get-Help [cmd]
```

Exercise 1: discovering the tracks

Different locomotives for different times; let's explore the two shells shipped by Windows.

```
Get-Help
Get-Command
```

Start the `cmd.exe` program. Type some commands you've learned the previous weeks on Linux. Now, start `Powershell` and repeat the experiment.

> Is `ls` always recognized? What is `ls` aliased to?

Similarly to Bash that we've used for the previous sessions, `cmd.exe` and `Powershell` come with custom commands and aliases. Today, we will focus on `Powershell`, as it is a handy tool to script on Windows.

Exercise 2: every object in the sky

Relevant cmdlets:

```
Get-ChildItem
Select-Object
```

The main difference between `Bash` and `Powershell` is in how they deal with input/outputs. `Bash` commands deal with strings of characters; they must be parsed, e.g., with `jq` to convert the string to an actual JSON object (see exercise 9 of the previous session). On the other hand, similarly to chaining coaches on a train, Powershell allows to chain commands while transferring **objects** from one to the other.

For instance, we can check which version of `Powershell` we're currently running using the following:

```
$PSVersionTable.PSVersion
```

Here, we've accessed the `PSVersion` property of the `$PSVersionTable` object.

> **a.** What other attributes/properties are available in the `$PSVersionTable` object?

Another way of accessing this information is via `Get-Host`.

> **b.** Using the pipe (`|`) operator and `Select-Object`, how can you select only the Version?

(Side note: check that you've opened the correct documentation version before proceeding.)

So far, we've only extracted a single value from a single object. In practice, cmdlets can manipulate multiple objects. Let's create a directory that will be used throughout the exercises (either manually through the File Explorer or via `New-Item -ItemType Directory`). It should not be empty; copy/move/create some files or directories inside it. Once this is done, navigate to it via Powershell.

> **c.** What cmdlet is used to change directories? If you've used `cd`, what is it aliased to?

List all possible properties of objects returned by `Get-ChildItem`. Hint: the wildcard character is the same for `Bash` and `Powershell`. Then, select only the FullName and CreationTime of said objects.

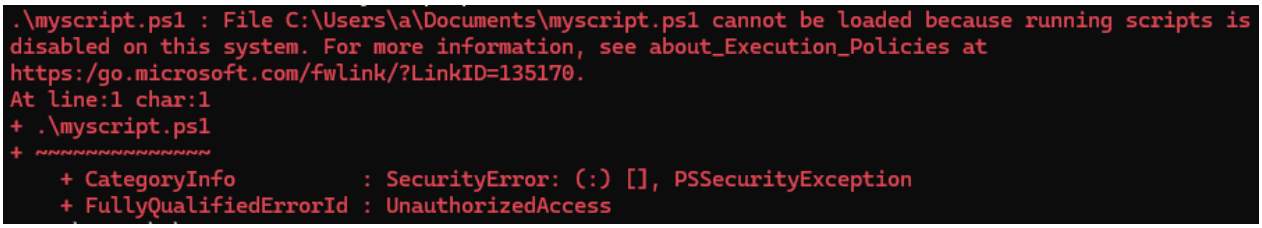
> **d.** What commands did you use?

Exercise 3: our first script

Keeping a train station clean requires more than typing single commands sequentially. Just as in Bash, we can create Powershell scripts that will run multiple commands one after the other. Let's create a simple script with the `ps1` extension and containing the following code illustrating two ways to print to the console:

```
Write-Host 'Hello world!'
'Hello world!' | Write-Host
```

It is possible you encounter an error similar to the following:



a. If we visit the linked website, what does "signed" mean? What does this protect against? (Remember slides 29 - 32 of lecture 3.)

To fix this issue, we can use the following command:

```
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
```

After running the command, you can check that the new execution policy is correctly applied using `Get-ExecutionPolicy`. It should return `RemoteSigned`.

Now that we're able to run our own scripts, we'll create a simple one that a) takes a filename as parameter, b) if the file exists, prints its size, c) if it does not, creates it and prints `$filename created`.

Relevant cmdlets:

```
Test-Path
Get-Item
Write-Host
New-Item

# To save the results of a given cmdlet, you can use $variables
# This is useful if you want to run a command once
# and use its output multiple times
$folderInfo = Get-Item -Path $folderPath
Write-Host "Folder Name: $($folderInfo.Name)"
Write-Host "Full Path:  $($folderInfo.FullName)"
Write-Host "Created On: $($folderInfo.CreationTime)"
```

b. Provide the resulting code.

Exercise 4: saving data and scheduling

A train engine is a complex system and we want to make sure that it does not overheat. Let's keep a close eye on the burning coal of processes.

Our goal is to a) list the processes currently running on the system, b) filter those using more than 100MB of RAM, and c) save the resulting list as a CSV file (i.e., comma separated values).

Relevant cmdlets:

```
Get-Process
Where-Object
Select-Object
Sort-Object
Export-Csv
```

It is possible the resulting CSV file will be empty if no process exceeds the RAM threshold. To test your program, you can momentarily reduce the value.

> **a.** Provide the resulting code.

Now, to better monitor our system, we would like to run the script every hour. Similarly to cron jobs on Linux, Windows and Powershell allows us to **schedule tasks**.

Relevant cmdlets:

```
New-ScheduledTaskAction
New-ScheduledTaskTrigger
Register-ScheduledTask
```

To manually run your scheduled task, you can use `Start-ScheduledTask -TaskName "MyScheduledTask"` with the name you previously defined. Similarly, you can check the latest time it ran with `Get-ScheduledTask -TaskName "MyScheduledTask" | Get-ScheduledTaskInfo`. A `LastTaskResult` equal to 0 means the task successfully ran. You can open the `Task Scheduler` to check that your task was correctly set.

Finally, there is no concept of an infinite task on Windows. Instead, we can schedule it to run for a **long time**, e.g., `-RepetitionDuration (New-TimeSpan -Days 9999)`. Note that `RepetitionInterval` uses the same syntax.

> **b.** Provide the commands used to set up the scheduled task.

Note: take care of relative and absolute paths defined to save the data as CSV. Using a relative path means the scheduled tasks will try to write in `C:\Windows\System32`, which won't work. You can specify the user's directory with an environment variable: `$env:USERPROFILE`.

> **c.** Open the Task Scheduler and check that your own scheduled task appears. Give an example of another scheduled task.

Do not forget to delete your scheduled task.

Exercise 5 (bonus): weather over the wire

As train driver, you'd like to know if it's going to rain in the following days. To do so, we can use a weather API, i.e., a website publishing data in a machine-readable format (here, JSON). For this exercise, we will use open-meteo.com.

The code should a) call the API endpoint with the correct latitude and longitude parameters to retrieve data, b) check for each date if it's going to rain, c) print a message containing the date and amount of rain, and d) if no rain is scheduled, print an information message.

Relevant cmdlets:

```
Invoke-RestMethod
# It is possible to use for loops in Powershell:
for ($i = 0; $i -lt $count; $i++) {
  # do something here
}
```

Provide the resulting code.

Improvement 1. Pass the latitude and longitude as parameters of the script.

Improvement 2. Using another API call, indicate if the following days were historically rainy.