1 Informationen für die Durcharbeitung des Dokuments

Dieses Dokument soll nach der von den Titel gegebenen Reihenfolge durchgearbeitet werden, allfällig erarbeitete Skripts sollen abgelegt werden. Bei späteren Übungen wird auf bereits erledigte Aufgaben zurückgegriffen und sie werden wieder verwendet.

Beim Erstellen der Skripts soll jede Zeile, welche einen Command ausführt mit einem Kommentar beschrieben / Erklärt werden. Dies dient zum Überprüfung des Verständnisses und zur Übung zum Schreiben von Kommentaren (im Kapitel „4 Kommentare in PowerShell“ wird das Kommentieren genauer beschrieben).

Tipp: Falls Befehle unklar sind, empfehle ich gerne https://ss64.com/ps/, da dort die meisten gängigen Befehle mit Beispielen sehr gut beschrieben werden.

2 Modulleitfaden

Dieses Dokument orientiert sich nach dem Schweizer Informatik-Modul 122 (Stand 2018).

Modulnummer	**122**
Titel	Abläufe mit einer Scriptsprache automatisieren
Kompetenz	Abläufe mit Scripts in der Systemadministration automatisieren.

Handlungsnotwendige Kenntnisse

1.1	Kennt Beispiele von Automatisierungsaufgaben.
1.2	Kennt Kriterien, die für die Beurteilung des Automatisierungspotentials von Bedeutung sind.
1.3	Kennt grundlegende Kontrollstrukturen und deren Einsatz bei der Ablaufautomatisierung.
1.4	Kennt das Vorgehen bei der Situationsanalyse und weiss, welchen Beitrag diese zur Definition eines klaren Auftrags liefert.
1.5	Kennt das Vorgehen zur grafischen Darstellung von Abläufen.
1.6	Kennt das Vorgehen zur Realisierung von Scripts in der Systemadministration.
2.1	Kennt grundlegende Funktionalitäten der eingesetzten Scriptsprache.
3.1	Kennt gängige Sicherheitsmassnahmen für den Einsatz von Scripts.
3.2	Kennt Integrationsmöglichkeiten von Scripts im eingesetzten Betriebssystem.
4.1	Kennt ein Testverfahren für Scripts.
4.2	Kennt das Testvorgehen zur Integration von Scripts.
5.1	Kennt die Elemente einer Dokumentation für die involvierten Rollen (z.B. System, Administrator, Entwickler)
5.2	Kennt die Bedeutung der Dokumentation in Bezug auf Qualitätssicherung und Wartbarkeit.

3 Grundlagen PowerShell

In diesem Kapitel werden die Grundlagen von PowerShell aufgezeigt.

3.1 Datentypen

In PowerShell nutzen wir folgende Datentypen, öffne zu den nachfolgenden Beispielen ein PowerShell-Fenster oder die PowerShell ISE, um die Beispiele direkt durchzuspielen. ***Dabei wird zum verstärkten Lernprozess empfohlen, die Beispiele abzutippen und nicht zu kopieren und einzufügen.***

Davon werden die für die Beispiele genutzten Datentypen genauer anhand eines Beispiels beschrieben, die restlichen werden lediglich aufgelistet.

3.1.1 INT / INT32

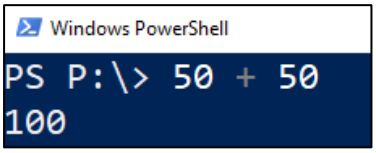
Unter einem INT-Wert, einem „integer“, verstehen wir eine Ganzzahl (z.B. 1, 2 oder 3), welche keine Fliesskommazahlen (Nachkommastellen) unterstützt. Dies bedeutet, dass es sich bei der Zahl Pi: „3.14“ um keinen INT-Wert handelt, da dies keine Ganzzahl ist.

In PowerShell werden meistens INT-Werte als Zahlenwerte genutzt. Zahlen mit Fliesskommazahlen kommen sehr selten zum Einsatz, aber existieren bei PowerShell als Datentyp „Float“ (siehe „4.1.5 Weitere Datentypen“).

Anwendungsbeispiel:

Eine klassische Anwendung von INT-Werten sind mathematische Berechnungen.

1. Öffne PowerShell
2. Gebe folgendes in das PowerShell-Fenster ein (ohne Anführungszeichen):
   ➔ „50 + 50“
3. Resultat: „100“



Zahlenwerte können so direkt zu Berechnungen verwendet werden.

3.1.2 String

Ein String ist eine Zeichenkette, wie z.B. das Wort „Baum“. Leerschläge, Zahlen und auch Sonderzeichen können im Datentypen String abgelegt werden.

Ein String kann in PowerShell mithilfe von Apostroph (‘) oder Anführungszeichen (“) deklariert werden:

➔ “Baum 123 @ Katze“
➔ ‘Baum 123 @ Katze‘

Anwendungsbeispiel 1: Ausgeben von Strings

1. Öffne PowerShell
2. Gebe folgendes in das PowerShell-Fenster ein:
   ➔ Write-Host „Baum 123 @ Katze“
3. Resultat: „Baum 123 @ Katze“
   ➔ Mithilfe des Commands „Write-Host“ oder kürzer „echo“ können Strings oder Zahlenwerte als String ausgegeben werden.

Anwendungsbeispiel 2: Zusammenhängen von Strings

1. Öffne PowerShell
2. Gebe folgendes in das PowerShell-Fenster ein:
   ➔ **"Das ist" + " " + "eine Zahl: " + 20**

3. Resultat: „Das ist eine Zahl: 20“
   ➔ Mithilfe eines Plus-Symbols können mehrere Strings oder auch **Zahlen** mit einem String – in einen neuen String zusammengefasst werden. Dabei muss beachtet werden, dass nicht automatisch **Leerzeichen** zwischen die Werte gefügt werden, diese müssen falls gewünscht manuell hinzugefügt werden.

```
PS P:\> "Das ist" +" " + "eine Zahl: " + 20
Das ist eine Zahl: 20
```

3.1.3 Array

Ein Array ist ein Datentyp, in welchem beliebig viele Zahlenwerte und Strings abgelegt werden können, welche man Array-Elemente oder Array-Items nennt.

Arrays können in PowerShell wie folgt deklariert werden, **mehrere Array-Elemente** werden mit einem **Komma getrennt**:

- Leeres Array: @()
- Array mit zwei String-Array-Elementen: @(“Arrayelement 1“ , “Arrayelement 2“)
- Array mit drei INT-Array-Elementen: @(20 , 30 , 40)
- Array mit gemischten Datentypen: @(„String-Element“ , 20)

Anwendungsbeispiel 1: Array-Deklaration

1. Öffne PowerShell
2. Gebe folgendes in das PowerShell-Fenster ein:
   ➔ @(“Arrayelement 1“, “Arrayelement 2“)

3. Resultat: Die einzelnen Elemente werden als eigene Zeile Ausgegeben
   ➔ *Arrayelement 1*
   ➔ *Arrayelement 2*

3.1.4 Boolean

Boolean ist z.Dt. als „Boolesche Variable“ bekannt. Dies ist ein Datentyp, welcher nur zwei bestimmte Werte akzeptiert – und zwar „TRUE“ (Wahr z.Dt.) oder „FALSE“ (Falsch z.Dt.).

Boolean werden genutzt, um zu definieren, ob etwas Wahr/True oder Falsch/False ist. Auf die Anwendung von Boolean-Variablen wird später eingegangen.

Anwendungsbeispiel True/False:

1. Öffne PowerShell
2. Gebe folgendes in das PowerShell-Fenster ein (ohne Klammerinfo):
   ➔ $true (oder $false)

3. Resultat: „True“ (oder „False“)

```
PS P:\> $true
True
PS P:\> $false
False
```

3.1.5 Weitere Datentypen

Bei PowerShell gibt es ausserdem noch folgende Datentypen, auf welche nicht weiter eingegangen werden und für die nachfolgenden Übungen nicht relevant sind:

Datatype	Description
[DateTime]	Date and time
[Guid]	Globally unique 32-byte identifier
[HashTable]	Hash table, collection of key-value pairs
[PsObject]	PowerShell object
[Regex]	Regular expression
[ScriptBlock]	PowerShell script block
[Float]	Floating point number
[Switch]	PowerShell switch parameter
[TimeSpan]	Time interval
[XmlDocument]	XML document

Quelle: https://4sysops.com/archives/the-powershell-variable-naming-value-data-type

3.2 Variablen

In Programmen braucht man eine Möglichkeit, um Werte zu speichern, z.B. Zwischenergebnisse von Berechnungen. Dies geschieht mit Hilfe von Variablen. Genau wie in der Mathematik ist eine Variable ein Platzhalter für einen Wert.

In PowerShell werden Variablen mit einem $-Zeichen deklariert, anschliessend kann mit einem Gleich-Symbol der Variable ein Wert zugewiesen werden:

$Test = „Das ist ein String“

In eine Variable kann jeder der folgenden beliebigen Typen abgelegt werden:

- Werte in einem beliebigen Datentyp
- Befehle
- Schleifen
- Inhalte anderer Variablen

Anwendungsbeispiel 1: String in einer Variable

1. Gebe folgendes in das PowerShell-Fenster ein:
   ➔ **$IchBinEineVariable = “das ist“ + “ “ + “ein String!“**
2. Gebe anschliessend folgendes in das PowerShell-Fenster ein:
   ➔ Write-Host $IchBinEineVariable
3. Resultat: „das ist ein String!“

Anwendungsbeispiel 2: Array in einer Variable

1. Gebe folgendes in das PowerShell-Fenster ein:
   ➔ **$IchBinEinArray = @(“Arrayelement 1“, “Arrayelement 2“)**
2. Gebe anschliessend folgendes in das PowerShell-Fenster ein:
   ➔ Write-Host $IchBinEinArray
3. Resultat: „Arrayelement 1 Arrayelement 2“
   ➔ Write-Host schreibt ein Array auf eine einzelne Zeile, eignet sich eher für Strings
   ➔ Einzelne Array-Elemente können mit deren Position im Array ausgegeben werden z.B. $IchBinEinArray[0] -> gibt Position 0 aus: „Arrayelement 1“

3.3 Bedingungen und Schleifen

3.3.1 Bedingungen

Eine Bedingung prüft ein Wert Variable mit den folgenden Operatoren:

Operator		Prüft, ob…
eq	**E**quals	zwei Werte übereinstimmen
ne	**N**ot **E**quals	zwei Werte nicht übereinstimmen
gt	**G**reater **T**han	der erste Wert grösser als der zweite ist
ge	**G**reater than or **E**quals To	der erste Wert grösser als oder gleich gross wie der zweite ist
lt	**L**esser **T**han	der erste Wert kleiner als der zweite ist
le	**L**esser than or **E**quals To	der erste Wert grösser als oder gleich gross wie der zweite ist

Anwendungsbeispiel:

1. Gebe folgendes in das PowerShell-Fenster ein:
   - 50 –gt 100

   Ausgabe: False

2. Gebe folgendes in das PowerShell-Fenster ein:
   - 50 -ge 50

   Ausgabe: True

Wie aus dem Beispiel ersichtlich gibt eine Bedingung eine Bool-Value zurück: True oder False – z.Dt. Wahr oder Falsch.

3.3.2 if / else - Konstrukt

Die if-Anweisung führt Anweisungen aus, wenn eine bestimmte Bedingung (s. letztes Thema „4.3.1 Bedingungen“) zu true ausgewertet wird. Wird die Bedingung zu false ausgewertet, können andere Anweisungen ausgeführt werden.

Der Syntax eines if/else Konstrukts ist wie folgt aufgebaut:

```
if(bedingung){
    # Falls zutrifft mach...
} else {
    # Falls nicht, mach...
}
```

Anwendungsbeispiel:

Empfohlen: Von diesem Punkt aus das Programm PowerShell ISE verwenden, da das Schreiben von Klammern in der PowerShell-Konsole sehr mühsam ist.

```
# Eine Zahl wird in die Variable $Zahl abgelegt
$Zahl = 50

# Prüfung, ob die Zahl grösser, als 100 ist
if($Zahl -gt 100){
    # Falls die Zahl grösser 100 ist:
    Write-Host "Die Zahl ist grösser als 100!"
} else {
    # Falls die Bedingung nicht zutrifft:
    Write-Host "Die Zahl ist kleiner als 100 / gleich 100!"
}
```

3.3.3 While-Schleife

Mit einer While-Schleife kann man eine Aktion durchführen, solange eine Kondition gültig ist.

Anwendungsbeispiel:

```
# Eine Zahl wird in die Variable $Zahl abgelegt
$Zahl = 50

# Solange die Zahl kleiner als 100 ist...
while($Zahl -lt 100){
    # Führe dies aus...
    $Zahl = $Zahl + 10
}
```

3.3.4 Foreach-Schleife

Mit einer Foreach-Schleife kann man für jedes Element in einem Array (s. Datentypen) den Vorgang in den Klammern ausführen.

Dies wird wie folgt aufgebaut, dabei kann die Variable `$Element` beliebig benannt werden und in der Iteration (Wiederholung) angesprochen werden. In der Variable $Element befindet sich der Wert des aktuellen Array-Elements während der Iteration:

```
# Eine Zahl wird in die Variable $Zahl abgelegt
$Array = @("Array-Element eins","Array-Element zwei","Array-Element drei")
$Zähler = 0

# Führe alle Commands innerhalb { und } mit jedem Array-Element aus...
foreach($Element in $Array){

    $String = "Dies ist das " + $Zähler + " Element im Array:"

    Write-Host $String

    $Element

    $Zähler = $Zähler + 1
}
```

3.3.5 Weitere Stichworte/Themen zu Schleifen in PowerShell

In PowerShell gibt es noch andere Commands/Schleifen im Bereich Schleifen, auf welche nicht genauer eingegangen werden:

- For
- Do-Until
- Continue
- Break

3.4 Pipe

Bei PowerShell gibt es die sehr hilfreiche Funktion „Pipe“, welche das Skripten wesentlich effizienter gestaltet. Diese Pipe Funktion wird durch ein **Pipe-Symbol ➔ |** (Alt Gr + 7) hinter einem Wert aufgerufen – und kann **den Wert** an einen **Befehl hinter der Pipe** weitergeben.

```
"test" | Write-Host
```

Ausgabe: „test“

- Der Write-Host Command erhält den String „test“ und wird ausgeführt.

Dies kann sehr effizient sein, als Beispiel eine Foreach-Schleife mithilfe der Pipe. Dabei kann mit der Variable $_ das weitergegebene Objekt angesprochen werden:

```
$Array | ForEach-Object { Write-Host $_ }
```

3.5 Funktionen

Eine Funktion ist zusammengefasster Script-Code, welcher als Befehl/Command aufgerufen werden kann, um so mehrfach verwendete Codeabschnitte nur einmal zu schreiben und aufzurufen. Dabei können Werte oder Variablen als Parameter mitgegeben werden, wie im Beispiel unterhalb als Variable $Parameter ersichtlich.

Beispiel 1: Funktion ohne Rückgabewert

```
#Funktion deklarieren
Function iteriereArray($Parameter){
    $Zähler = 0
    Write-Host "Iteration"
    Write-Host "--------------------------------"
    foreach($Element in $Parameter){
        $String = "Dies ist das " + $Zähler + " Element im Array:"
        Write-Host $String
        Write-Host $Element
        $Zähler = $Zähler + 1
    }
    Write-Host "--------------------------------"
}


$ErstesArray = @("Array-Element eins","Array-Element zwei","Array-Element drei")
$ZweitesArray = @("Array-Element eins","Array-Element zwei","Array-Element drei")

# Funktionsaufrufe
iteriereArray $ErstesArray
iteriereArray $ZweitesArray
```

Beispiel 2: Funktion mit Rückgabewert

Mithilfe von „return“ kann in einer Funktion ein Wert zurückgegeben werden.

```
#Funktion deklarieren
Function verdoppleFallsMehrAlsHundert($Parameter){
    if($Parameter -gt 100) {
        return $Parameter * 2
    } else {
        return $false
    }
}

verdoppleFallsMehrAlsHundert 250

verdoppleFallsMehrAlsHundert 100
```

4 Kommentare in PowerShell

In PowerShell gibt es einige verbreiteten Wege, den Code zu kommentieren. Nachfolgend werden die gängigsten beschrieben. Es wird zu Übungszwecken empfohlen, möglichst viele Zeile mit einem Kommentarblock zu versehen, um sicherzugehen, dass der Code verstanden wurde.

4.1 Kommentare Generell

In PowerShell können Kommentare wie folgt erstellt werden:

```
# Dies ist ein Kommentar

<#
   Dies ist ein
   mehrzeiliger
   Kommentar
#>
```

Es macht ebenfalls Sinn der Skriptstart und Funktionen wie in die nächsten zwei Kapitel beschrieben zu kommentieren, dies geht leider oft vergessen.

4.2 Kommentar am Anfang eines Skripts

Es ist gängig, dass am Anfang des Skripts ein grösserer Kommentarblock zur Erklärung des Scripts genutzt wird:

```
# Scriptname: GUI_Computerinformation.ps1
<#
.SYNOPSIS
    The Computer information GUI is ran automatically by GUI_MAIN.ps1

.PARAMETER
    (Hostname), (SAM-Account-Name)

.NOTES
    Author:         Alejandro Probst
    Version:        1.0
    Date v1.0:      17.04.2019
    ____________________________________________________

    Changelog:
    17.04.2019    Alejandro Probst    Version 1.0
#>
```

4.3 Kommentar am Anfang einer einer Funktion

Auch vor Funktionen werden meistens deren Nutzen und der Rückgabewert beschrieben, falls die Funktion einen Wert zurückgibt:

```
<#
    .DESCRIPTION
    This function runs a Query on the Database XYZ

    .PARAMETER
    Query(string)

    .OUTPUT
    Returns the SQL-Result as an Array/Object[]
#>
```

5 Erstellen des ersten Skripts: „setPassword“

In diesem Kapitel wird das Skript „setPassword.ps1“ erstellt.

5.1 Grundlage

Erstelle ein Skript, welches den Nutzer nach einem Passwort fragt. Anschliessend soll das Skript nach einer Bestätigung des Passworts fragen.

Mit diesen Informationen soll das Skript prüfen, ob zwei Mal dasselbe Passwort eingegeben wurde. Falls dies der Fall ist, soll das Skript beendet werden, ansonsten soll das Skript erneut nach dem Passwort und der Wiederholung fragen.

Tipps:

- Probiere den Command „Read-Host“ aus:
```
$Nutzereingabe = Read-Host "Geben Sie das Passwort ein"
```
- Schaue dir das Kapitel 4.3.1 an, falls du bei der Überprüfung nicht weiterkommst
- Schaue dir das Kapitel 4.3.3 an, falls du nicht weisst, wie du bei einer Fehleingabe erneut nach dem Passwort fragen kannst

5.2 Zusatz

Frage den Nutzer, ob er ein Passwort generieren oder setzen möchte. Erstelle anschliessend einen Passwort-Generator, welcher ein Passwort mit einer **Zahl**, einem **Sonderzeichen** und einem **Wort** generiert.

Diese **drei Teile** des Passworts sollen zufällig generiert werden, sodass der Generator nicht immer dieselben Zahlen, Sonderzeichen oder Wörter ausgibt.

Am Ende soll der Generator ein Passwort als Rückgabewert ausgeben.

Tipps:

- Probiere den Command „Get-Random“ aus:
```
Get-Random -Maximum 100 -Minimum 1
```
- Schaue dir zum Zusammensetzen des Passworts das zweite Anwendungsbeispiel im Kapitel 4.1.2 genauer an, kombiniere dies mit Arrays (Kapitel 4.1.3 Beispiele nochmals durchlesen)

6 Regex in PowerShell: „inputValidator“

In diesem Kapitel wird das Skript „inputValidator.ps1“ erstellt.

6.1 Grundlage

Erstelle ein Skript, welches den Nutzer nach den folgenden Informationen fragt:

- Name und Vorname
- Adresse (Strasse und Nummer)
- Postleitzahl und Ort
- Telefonnummer

Anschliessend soll das Skript die eingegebenen Angaben mittels Regex prüfen. Finde mithilfe der Seite https://regex101.com/ heraus, wie Regex funktioniert und lies unter folgendem Link nach, wie man einen Regex in PowerShell nutzt: https://ss64.com/ps/syntax-regex.html

- Tipp: Regex muss nicht wie in anderen Sprachen importiert werden, sondern gehört zur Basis von PowerShell

Falls eine Eingabe nicht der Anforderung entspricht, soll das Skript den Nutzer erneut nach der inkorrekten Angabe fragen. Anschliessend sollen die Informationen Zusammengefasst und in der Konsole ausgegeben werden.

6.2 Zusatz

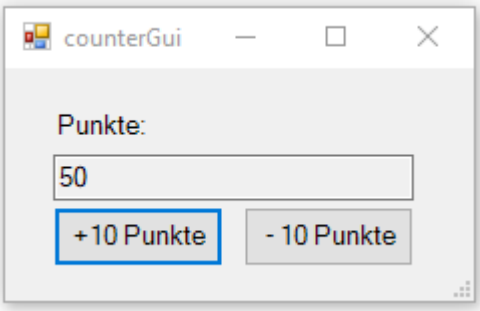
Frage den Nutzer zusätzlich nach seiner E-Mail-Adresse und prüfe diese ebenfalls mit einem Regex.

7 Erstes PowerShell-GUI: „counterGui“

In diesem Kapitel wird das Skript „counterGui.ps1“ erstellt.

7.1 Grundlage

Erstelle eine **Grafische Nutzeroberfläche** (Graphical User Interface = **GUI**). Kopiere/Erstelle diese analog der unterhalb ersichtlichen Abbildung:



Du kannst dazu den folgenden Online-Editor nutzen, oder mit einer Internetrecherche das GUI selbst erstellen: https://poshgui.com/Editor

Tipps:

- Die ersichtlichen GUI-Elemente nennt man; Form, Label, Textfeld und Button
- Mit Poshgui kann man über den Code-Button das GUI in PowerShell-Code generieren und in das Skript in der ISE einfügen.
- Falls du den Editor nutzt: Kommentiere den generierten Code damit du jede Zeile verstehst
- Das Objekt „system.Windows.Forms.Form“ verfügt über die Funktion ShowDialog(). Diese wird zwingend benötigt, um das GUI anzuzeigen und kann wie folgt aufgerufen werden, wobei $form1 durch die Variable der Form ersetzt werden muss:

```
$form1.ShowDialog()
```

7.2 Zusatz Level 1

1. Das Textfeld soll nicht bearbeitbar sein und zu Beginn den Wert „50“ beinhalten
2. Beim Knopfdruck auf „+10 Punkte“ sollen 10 Punkte zum Wert im Textfeld hinzugefügt werden.
3. Beim Knopfdruck auf „-10 Punkte“ sollen 10 Punkte vom Wert im Textfeld abgezogen werden. Dabei soll der Button deaktiviert werden, wenn das Textfeld auf 0 Punkt fällt, damit die Zahl nicht ins Minus fällt.

➔ **Tipp: mit [int] vor einer Variable kann diese als Integer-Wert ausgegeben werden!**

7.3 Zusatz Level 2

1. Ändere Hintergrundfarben, Schriftarten und Schriftgrössen der Elemente im GUI beliebig.
2. Erstelle einen „-1 Punkt“ und „+1 Punkt“ Button und passe die Form entsprechend an. Stelle dabei sicher, dass der Wert beim Knopfdruck auf „+10 Punkte“ nicht unter 0 fällt

8 Flussdiagramme

Erstelle zu den aus den in den Kapiteln 6, 7 und 8 geschrieben Skripts je ein Flussdiagramm, welches den Ablauf der Skripts erklärt.

Mache dich im Internet über den Begriff „Entity-Relationship-Modell“ schlau und erkläre die Anwendung und die einzelnen Komponenten eines ERM.