PowerTips MONTHLY

Part of the PowerShell.com reference library, brought to you by **Dr. Tobias Weltner**

Volume 3 | August 2013

This Month's Topic:

Date, Time, and Culture

Dr. Tobias Weltner

Sponsored by idera® Application & Server Management

Table of Contents

1. Getting Weekdays and Current Month
2. Filtering Day of Week
3. Display Work Hours in Prompt
4. Calculating Time Differences
5. Using Relative Dates
6. Finding Out About DateTime Methods
7. Secret Time Span Shortcuts
8. Getting Short Dates
9. Finding Leap Years
10. Finding Days in Month
11. Calculate Time Zones
12. Converting UNIX Time
13. Counting Work Days
14. Getting Time Zones
15. Parsing Date and Time
16. Using Culture-Neutral Date and Time Formats
17. Get Localized Day Names
18. Lunch Time Alert
19. Testing Numbers and Date
20. Adding Clock to PowerShell Console
21. Converting User Input to Date
22. Using Cultures
23. Listing Available Culture IDs
24. Translating Culture IDs to Country Names

1. Getting Weekdays and Current Month

Get-Date can extract valuable information from dates. All you need to know is the placeholder for the date part you are after. Then, repeat that placeholder to see different representations. For example, an upper-case “M” represents the month part:

```
Get-Date -Format ‘M’
Get-Date -Format ‘MM’
Get-Date -Format ‘MMM’
Get-Date -Format ‘MMMM’
```

Here are the most common placeholders (case-sensitive):

Day	d
Month	M
Year	y
Hour (12hr)	h
Hour (24hr)	H
Minute	m
Second	s

Sponsored by idera® Application & Server Management

2. Filtering Day of Week

To check whether the current date is a weekend date, use a function like this:

```
function Test-Weekend
{
  (Get-Date).DayOfWeek -gt 5
}
```

You can use it like this:

```
If (Test-Weekend) {
  'No service at weekends'
}
Else
{
  'Time to work'
}
```

3. Display Work Hours in Prompt

You can add date and time information into your PowerShell prompt. This prompt function displays the minutes (or hours) you have been working so far, and show the current path in the console title bar instead:

```
function prompt {
  $work = [Int] ((New-TimeSpan (Get-Process -Id $pid).StartTime).TotalMinutes)
  "$work min> "
  $host.UI.RawUI.WindowTitle = (Get-Location)
}
```

This version displays hours:

```
function prompt {
  $work = [Int] ((New-TimeSpan (Get-Process -Id $pid).StartTime).TotalHours)
  '{0:0.0}h > ' -f $work
  $host.UI.RawUI.WindowTitle = (Get-Location)
}
```

4. Calculating Time Differences

Use New-TimeSpan and submit a date if you'd like to know how long it is until next Christmas. New-TimeSpan automatically compares this date to the current date (and time) and returns the time span in a number of different formats from which you can choose.

For example, to get the time span in days, use this:

```
(New-TimeSpan 2013-12-25).Days
```

Note that future dates return negative time spans because by default, PowerShell binds the argument to -Start instead of -End. Explicitly bind your date to -End will fix this:

```
$days = (New-TimeSpan -End 2013-12-25).Days
"$days days to go until next Christmas"
```

Author Bio

Tobias Weltner is a long-term Microsoft PowerShell MVP, located in Germany. Weltner offers entry-level and advanced PowerShell classes throughout Europe, targeting mid- to large-sized enterprises. He just organized the first German PowerShell Community conference which was a great success and will be repeated next year (more on www.pscommunity.de).. His latest 950-page "PowerShell 3.0 Workshop" was recently released by Microsoft Press.

To find out more about public and in-house training, get in touch with him at tobias.weltner@email.de.

5. Using Relative Dates

Maybe you want to find files older than 14 days, or pull event log entries written in the past 24 hours. This is when you need relative dates. You create relative dates by subtracting a given amount of time (a time span) from the current date.

This will get you the date 2 days ago:

```
(Get-Date) - (New-TimeSpan -Days 2)
```

You get the same information by using the DateTime methods:

```
(Get-Date).AddDays(-2)
```

And this would pull all error events from the System log within the past 12 hours:

```
Get-EventLog -LogName System -EntryType Error -After (Get-Date).AddHours(-12)
```

6. Finding Out About DateTime Methods

To create relative dates like “10 days ago,” there are two paths. Administrators often add or remove time spans created by New-TimeSpan:

```
(Get-Date) - (New-TimeSpan -Days 10)
```

Developers prefer object methods:

```
(Get-Date).AddDays(-10)
```

To use .NET DateTime methods, you will need to know them first. The PowerShell ISE editor provides IntelliSense, and you can also ask PowerShell directly:

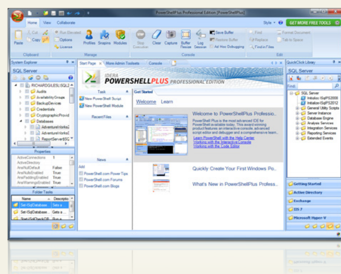
```
PS> Get-Date | Get-Member -MemberType *Method


   TypeName: System.DateTime

Name                 MemberType Definition
----                 ---------- ----------
Add                  Method     datetime Add(timespan value)
AddDays              Method     datetime AddDays(double value)
AddHours             Method     datetime AddHours(double value)
AddMilliseconds      Method     datetime AddMilliseconds(double value)
AddMinutes           Method     datetime AddMinutes(double value)
AddMonths            Method     datetime AddMonths(int months)
AddSeconds           Method     datetime AddSeconds(double value)
AddTicks             Method     datetime AddTicks(long value)
AddYears             Method     datetime AddYears(int value)
(...)
```

This will list all methods provided by DateTime objects. Thus, to add six years to today’s date, you can use this:

```
(Get-Date).AddYears(6)
```

Powershell Plus

FREE TOOL TO LEARN AND MASTER POWERSHELL FAST

- Learn PowerShell fast with the interactive learning center
- Execute PowerShell quickly and accurately with a Windows UI console
- Access, organize and share pre-loaded scripts from the QuickClick™ library
- Code & Debug PowerShell 10X faster with the advanced script editor

7. Secret Time Span Shortcuts

Usually, to create time spans, you will use New-TimeSpan. However, you can also use a more developer-centric approach by converting a number to a time span type:

```
[TimeSpan]100
```

This will get you a time span of 100 ticks, which is the smallest unit available to measure time. As PowerShell does support some hidden shortcuts for other units, this will create a time span of exactly five days:

```
[TimeSpan]5d
```

8. Getting Short Dates

Objects contain useful methods to access the object data. For example, DateTime objects support methods to display the date and time in various formats.

Use this if you just need a short date:

```
(Get-Date).ToShortDateString()
```

9. Finding Leap Years

You will find that the DateTime type supports a number of static methods to check dates. For example, you can check whether a year is a leap year like this

```
[DateTime]::IsLeapYear(1904)
```

10. Finding Days in Month

If you need to determine the days in a given month, you can use the static DaysInMonth() function provided by the DateTime type. As you can see, the days in a month are not always the same for every year:

```
PS> [DateTime]::DaysInMonth(2013, 2)
28

PS> [DateTime]::DaysInMonth(2014, 2)
28

PS> [DateTime]::DaysInMonth(2016, 2)
29
```

11. Calculate Time Zones

If you need to find out the time in another time zone, you can convert your local time to Universal Time and then add the number of offset hours to the time zone you want to display:

```
((Get-Date).ToUniversalTime()).AddHours(8)
```

Technical Editor Bio

Aleksandar Nikolic, Microsoft MVP for Windows PowerShell, a frequent speaker at the conferences (Microsoft Sinergija, PowerShell Deep Dive, NYC Techstravaganza, KulenDayz, PowerShell Summit) and the cofounder and editor of the PowerShell Magazine (http://powershellmagazine.com). He is also available for one-on-one online PowerShell trainings. You can find him on Twitter: https://twitter.com/alexandair

12. Converting UNIX Time

Some values in the Windows Registry are formatted using UNIX datetime format:

```
$Path = 'HKLM:\Software\Microsoft\Windows NT\CurrentVersion'

Get-ItemProperty -Path $Path |
  Select-Object -ExpandProperty InstallDate
```

The InstallDate value returns a large number. In UNIX format, dates and times are expressed in seconds passed since 1/1/1970.

Here is a little function that converts this back to real dates:

```
function ConvertFrom-UnixTime {
  param(
      [Parameter(Mandatory=$true, ValueFromPipeline=$true)]
    [Int32]
    $UnixTime
  )
  begin {
    $startdate = Get-Date -Date '01/01/1970'
  }
  process {
    $timespan = New-TimeSpan -Seconds $UnixTime
    $startdate + $timespan
  }
}
```

So now, you can correctly read the installation date from the Registry:

```
$Path = 'HKLM:\Software\Microsoft\Windows NT\CurrentVersion'

Get-ItemProperty -Path $Path |
  Select-Object -ExpandProperty InstallDate |
  ConvertFrom-UnixTime
```

If you wanted to know the number of days since your copy of Windows was installed, calculate the time difference:

```
$Path = 'HKLM:\Software\Microsoft\Windows NT\CurrentVersion'

Get-ItemProperty -Path $Path |
  Select-Object -ExpandProperty InstallDate |
  ConvertFrom-UnixTime |
  New-TimeSpan |
  Select-Object -ExpandProperty Days
```

13. Counting Work Days

Ever wanted to know how many days you need to work this month? Here's a clever function that lists all the weekdays you want that are in a month. By default, Get-WeekDay returns all Mondays through Fridays in the current month. You can specify different months (and years), too, if you want to.

```
function Get-Weekday {
    param(
        $Month = $(Get-Date -Format 'MM'),
        $Year = $(Get-Date -Format 'yyyy'),
        $Days = 1..5
    )

    $MaxDays = [System.DateTime]::DaysInMonth($Year, $Month)

    1..$MaxDays | ForEach-Object {
        Get-Date -Day $_ -Month $Month -Year $Year |
          Where-Object { $Days -contains $_.DayOfWeek }
    }
}
```

Pipe the result to Measure-Object to count the days.

```
PS> Get-Weekday | Measure-Object | Select-Object -ExpandProperty Count
20
```

You can even pick the weekdays you want to get included. The weekday ID starts with 0 (Sunday) and ends with 6 (Saturday). So this gets you all the weekends in the current month:

```
PS> Get-Weekday -Days 'Saturday', 'Sunday' | Measure-Object | Select-Object -ExpandProperty Count
10
```

And if you know each Monday is a soccer meeting, this will tell you how many Mondays there are this month:

```
PS> Get-Weekday -Days 'Monday'
```

14. Getting Time Zones

Here's a low-level call that returns all time zones:

```
PS> [System.TimeZoneInfo]::GetSystemTimeZones()


Id                         : Dateline Standard Time
DisplayName                : (UTC-12:00) International Date Line West
StandardName               : Dateline Standard Time
DaylightName               : Dateline Daylight Time
BaseUtcOffset              : -12:00:00
SupportsDaylightSavingTime : False

Id                         : UTC-11
DisplayName                : (UTC-11:00) Coordinated Universal Time-11
StandardName               : UTC-11
DaylightName               : UTC-11
BaseUtcOffset              : -11:00:00
SupportsDaylightSavingTime : False

Id                         : Hawaiian Standard Time
DisplayName                : (UTC-10:00) Hawaii
StandardName               : Hawaiian Standard Time
DaylightName               : Hawaiian Daylight Time
BaseUtcOffset              : -10:00:00
SupportsDaylightSavingTime : False

(...)
```

To find the time zone you are in, use this:

```
PS> [System.TimeZoneInfo]::Local


Id                         : W. Europe Standard Time
DisplayName                : (UTC+01:00) Amsterdam, Berlin, Bern, Rome, Stockholm, Vienna
StandardName               : W. Europe Standard Time
DaylightName               : W. Europe Daylight Time
BaseUtcOffset              : 01:00:00
SupportsDaylightSavingTime : True



PS> ([System.TimeZoneInfo]::Local).StandardName
W. Europe Standard Time
```

15. Parsing Date and Time

Parsing a date and/or time information is tricky because formatting depends on the regional settings. This is why PowerShell can convert date and time based on your regional settings or in a culture-neutral format. Let's assume this date:

```
PS> $date = '1/6/2013'
```

If you convert this to a DateTime type, PowerShell always uses the culture-neutral format (US format), regardless of your regional settings. The output is shown here on a German system:

```
PS> $date = '1/6/2013'
PS> [DateTime]$date
Sonntag, 6. Januar 2013 00:00:00
```

To use your regional DateTime format, use the Parse() method which is part of the DateTime type, like this:

```
PS> [DateTime]::Parse($date)
Samstag, 1. Juni 2013 00:00:00
```

Alternately, you can use Get-Date and the -Date parameter:

```
PS> [DateTime]::Parse($date)
Samstag, 1. Juni 2013 00:00:00
```

16. Using Culture-Neutral Date and Time Formats

Using regional date and time formats can produce a lot of problems because hard-coded date and time formats in your script may break once the script runs in a different culture.

That's why you should always use a special date and time format that works in all cultures equally well. It looks like this: yyyy-MMdd HH:mm:ss.

So, to convert Oct 2, 2014 at 7:34 in the morning into a valid DateTime object, try this format:

```
PS> $format = '2014-10-02 07:34:00'

PS> [DateTime]$format
Donnerstag, 2. Oktober 2014 07:34:00


PS> [DateTime]::Parse($format)
Donnerstag, 2. Oktober 2014 07:34:00


PS> Get-Date -Date $format
Donnerstag, 2. Oktober 2014 07:34:00
```

As you can see, the date and time was converted correctly no matter what conversion logic you used. Likewise, the format will work equally well in different cultures.

17. Get Localized Day Names

To get a list of day names, you can use this line:

```
PS> [System.Enum]::GetNames([System.DayOfWeek])
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
```

However, this returns a culture-neutral list which is not returning the day names in a localized (regional) form. To get the localized day names, use this line instead:

```
PS> 0..6 | ForEach-Object {
  [Globalization.DatetimeFormatInfo]::CurrentInfo.DayNames[$_]
}

Sonntag
Montag
Dienstag
Mittwoch
Donnerstag
Freitag
Samstag
```

To get localized month names instead, try this:

```
PS> 0..11 | ForEach-Object {
  [Globalization.DatetimeFormatInfo]::CurrentInfo.MonthNames[$_]
}

Januar
Februar
März
April
Mai
Juni
Juli
August
September
Oktober
November
Dezember
```

18. Lunch Time Alert

Here's a fun prompt function that turns your input prompt into a short prompt and displays the current path in your PowerShell window title bar. In addition, it has a lunch count down, displaying the minutes to go. Three minutes before lunch time, the prompt also emits a beep tone so you never miss the perfect time for lunch anymore.

```
function prompt {
    $lunchtime = Get-Date -Hour 12 -Minute 30
    $timespan = New-TimeSpan -End $lunchtime
    [Int]$minutes = $timespan.TotalMinutes
    switch ($minutes) {
        { $_ -lt 0 }   { $text = 'Lunch is over. {0}'; continue }
        { $_ -lt 3 }   { $text = 'Prepare for lunch!  {0}' }
        default        { $text = '{1} minutes to go... {0}' }
    }

    'PS> '
    $Host.UI.RawUI.WindowTitle = $text -f (Get-Location), $minutes
    if ($minutes -lt 3) { [System.Console]::Beep() }
}
```

19. Testing Numbers and Date

With a bit of creativity (and the help from the -as operator), you can create powerful test functions. These two test for valid numbers and valid DateTime information:

```
function Test-Numeric($Value) { ($Value -as [Int64]) -ne $null }
function Test-Date($Value) { ($Value -as [DateTime]) -ne $null }
```

If things get more complex, you can combine tests. This one tests for valid IP addresses:

```
function Test-IPAddress($Value) { ($Value -as [System.Net.IPAddress]) -ne $null -and ($Value -as
[Int]) -eq $null }
```

The limit here is the type converter. Sometimes, you may be surprised what kind of values are convertible into a specific type. The date check, on the other hand, illustrates why this can be very useful. Most cmdlets that require dates, do support the same type conversion rules.

20. Adding Clock to PowerShell Console

Maybe you'd like to include dynamic information such as the current time into the title bar of your PowerShell console. You could update the console title bar inside your prompt function, but then the title bar would only get updated each time you press ENTER. Also, the prompt function may be overridden from other code.

A better way is to spawn another PowerShell thread and let it update the title bar text, preserving whatever has been written to the title bar and just adding the information you want, for example the current time (or, for that matter, the current battery load or CPU load, RSS feed or whatever you want to have an eye on).

Here's the code. Be aware that this will only work inside a real PowerShell console, not inside the ISE editor. After you run this code, call Add-Clock to add the current time to your console title bar.

```
function Add-Clock {
 $code = {
    $pattern = '\d{2}:\d{2}:\d{2}'
    do {
      $clock = Get-Date -Format 'HH:mm:ss'

      $oldtitle = [system.console]::Title
      if ($oldtitle -match $pattern) {
        $newtitle = $oldtitle -replace $pattern, $clock
      } else {
        $newtitle = "$clock $oldtitle"
      }
      [System.Console]::Title = $newtitle
      Start-Sleep -Seconds 1
    } while ($true)
  }
```

```
  $ps = [PowerShell]::Create()
  $null = $ps.AddScript($code)
  $ps.BeginInvoke()
}

Add-Clock
```

21. Converting User Input to Date

PowerShell uses the US/English date format when converting user input to DateTime, which can cause unexpected results if using a different culture.

For example, on German systems "1.3.2014" typically resolves to March 1, 2014. PowerShell converts this to January 3, 2014, though.

To convert DateTime values based on your current culture, use the DateTime's Parse() method:

```
[DateTime]::Parse('1.3.2014')
```

To handle invalid data entered by a user, there are two workarounds. First, you can use -as to see if the user input can be converted to a DateTime at all. If so, use Parse() to get the culture-specific date:

```
$date = Read-Host 'Enter your birthday'
if (($date -as [DateTime]) -ne $null)
{
  $date = [DateTime]::Parse($date)
  $date
}
else
{
 'You did not enter a valid date!'
}
```

Or, you can use an error handler to catch the exception like so:

```
$date = Read-Host 'Enter your birthday'
try
{
  $date = [DateTime]::Parse($date)
  $date
}
catch
{
  'You did not enter a valid date!'
}
```

22. Using Cultures

Since PowerShell is culture-independent, you can pick any culture you want and use the culture-specific formats. The following script instantiates the Japanese culture, outputting a number as currency first in your current culture and then in the Japanese culture.

```
PS> $culture = New-Object System.Globalization.CultureInfo("ja-JP")


PS> $number = 100


PS> $number.ToString('c')

100,00 €

PS> $number.ToString('c', $culture)

¥100
```

Learn more about what cultures are available and their identifiers by visiting MSDN:

http://msdn.microsoft.com/en-us/library/system.globalization.cultureinfo.aspx

The ToString() method is available for all objects and can be used for other data types as well:

http://msdn.microsoft.com/en-us/library/fbxft59x(VS.80).aspx

Here is an example with dates:

```
PS> (Get-Date).ToString()
25.06.2013 14:46:28

PS> (Get-Date).ToString('d', $culture)
2013/06/25

PS> (Get-Date).ToString($culture)
2013/06/25 14:46:43
```

23. Listing Available Culture IDs

Ever needed to get your hands on some specific culture IDs? Here is how you can create a list:

```
PS> [System.Globalization.CultureInfo]::GetCultures('InstalledWin32Cultures')

LCID             Name             DisplayName
----             ----             -----------
1                ar               Arabic
2                bg               Bulgarian
3                ca               Catalan
4                zh-Hans          Chinese (Simplified)
5                cs               Czech
6                da               Danish
(...)
```

Next, you can use a culture to output information formatted in that culture.

Either, you change the thread culture and add the command to the line that changed the culture. The change is valid only for the line that changes the culture:

```
PS> $c = [System.Globalization.CultureInfo]'ar-IQ'

PS> Get-Date

Dienstag, 25. Juni 2013 14:49:43


PS> [System.Threading.Thread]::CurrentThread.CurrentCulture = $c; Get-Date

25 حزيران, 2013 02:49:53 م


PS> Get-Date

Dienstag, 25. Juni 2013 14:49:59
```

Sponsored by idera® Application & Server Management

Or, you use ToString() and submit the culture you want to use for display:

```
PS> (Get-Date).ToString($c)
25/06/2013 02:50:10 م

PS> (Get-Date).ToString('MMM', $c)
حزيران
```

Note: the console character set is not able to display Chinese or Arabian characters. You may want to run that command inside the PowerShell ISE or another Unicode-enabled environment.

24. Translating Culture IDs to Country Names

Ever wondered what a specific culture ID stands for? Here is a clever function that will translate a culture ID to the full culture name:

```
PS> [System.Globalization.CultureInfo]::GetCultureInfoByIetfLanguageTag('de-DE')

LCID             Name             DisplayName
----             ----             -----------
1031             de-DE            German (Germany)
```