A Python Developer's Guide to Powershell

May 16, 2013 Mohamed Akram

Introduction

Python is a great scripting language - it’s available by default on Linux and Mac and so it’s easy to quickly write a short script that runs on many systems. However, this isn’t the case on Windows. You need to install Python or wrap your application to distribute it. Sometimes this is inconvenient, especially if you want to do something simple or deal directly with Windows specific functions, so we need an alternative. This is where PowerShell comes in.

Getting started

PowerShell is a very powerful scripting language and is cmd.exe’s successor. It also comes with a capable IDE which you can use for this tutorial. Find tʼ ȩŐǨŗ ỹ ẎĆŗ ỹ ǴŵĎḎǴḵkừĎc in the Start menu or open tʼ ȩŐǨŗ ỹ ẎĆŗ ỹ ǴŵĎḎǴḵkand type the `ise` command to start it. Now let’s write our first program.

In Python, this is how you’d write a “hello world” program:

```python
print("hello, world")
```

It’s even easier in PowerShell:

```powershell
"hello, world"
```

PowerShell automatically outputs any strings to the screen without any command. That’s kind of cheating, so here’s the explicit way to do it:

```
Write-Host "hello, world"
```

The names of PowerShell ǜN̂ ǨḴǴẤY are very consistent. They follow what Microsoft calls a verb-noun pair convention.

Variables

Variables are defined by prepending a dollar sign to the variable name.

```
$a = 5
$b = 6
```

You can also do Python style variable swapping:

```
$a, $b = $b, $a

Write-Host a=$a, b=$b
Write-Host ($a + $b)
```

Without parentheses, the second `Write-Host` command would produce `5 + 6` instead of `11`.

PowerShell also supports arrays with mixed types,

```
$Array = 2, "cheese", 6.5, "cake"

# Explicit syntax
$Array = @(5, "ice", 3.14, "cream")

# Inclusive range
$Array = (1..10)
```

It also supports dictionaries:

```
$Table = @{"a" = "apple"; "b" = "ball"}
$Table["a"] = "acorn"

# Loop through keys
foreach ($k in $Table.keys) {
    Write-Host $k, $table[$k]
}
```

Loops

What if we wanted to print the numbers from 1 to 10. In Python, it would look like this:

```
for i in range(1, 11):
    print(i)
```

Similarly, in PowerShell:

```
foreach ($i in (1..10)) {
    Write-Host $i
}
```

You could just as easily use a while loop:

```
$i = 1
while ($i -le 10) {
    Write-Host $i
    # Increment i
    $i++
}
```

Another interesting loop method is do...until:

```
$i = 0
do {
    $i++
    Write-Host $i
} until ($i -eq 10)
```

Conditions

Conditions look slightly different in PowerShell:

```
# Equal to
$True -eq $False # False
# Not equal to
$True -ne $False # True

# Less than
5 -lt 10 # True
# Greater than
5 -gt 10 # False

# Less than or equal to
5 -le 10 # True
# Greater than or equal to
5 -ge 10 # False
```

PowerShell supports the same logical operators as Python including `-and` , `-or` , and `-not` .

```
# Logical operators
$Happy = $True
$KnowIt = $True

if ($Happy -and $KnowIt) {
    "Clap hands!"
}
```

Functions

A typical Fibonacci function in Python looks like this:

```
def fib(n):
    if n < 2:
        return n
    return fib(n - 2) + fib(n - 1)
```

To define a function in PowerShell, use the `Function` keyword:

```
Function Fib($n) {
    if ($n -lt 2) {
        return $n
    }
    return (Fib($n - 2)) + (Fib($n - 1))
}
```

Again, the parentheses around the `Fib` calls are important to properly return a value.

List comprehensions

To get a list of all the multiples of 2 from 1 to 20, you'd do this in Python:

```
multiples = [i for i in range(1, 21) if i % 2 ==
```

PowerShell:

```
$Multiples = 1..20 | Where-Object {$_ % 2 -eq 0}
```

The list comprehensions are somewhat different than in Python and make use of piping, which is a very powerful tool in PowerShell. In this case, a range from 1 to 20 is piped to the `Where-Object` command which filters the list of items according to the condition `$_ % 2 -eq 0`. The `$_` variable essentially refers to each item in a list of objects.

To do an operation on each multiple of two, say find its square, we pipe the multiples to the `ForEach-Object`:

```
$Squares = $Multiples | ForEach-Object {$_ * $_}
```

You can also do it all on one line:

```
$Squares = 1..20 | ? {$_ % 2 -eq 0} | % {$_ * $_
```

The `?` is an alias for `Where-Object` and `%` is an alias for `ForEach-Object`.

Example program

I've written a short program in both Python and PowerShell that downloads a bunch of xkcd comics to a "xkcd" folder on the Desktop.

Python:

```
import os
from urllib.request import urlopen
from html.parser import HTMLParser


# Parse xkcd page
class Parser(HTMLParser):
    def __init__(self):
        super().__init__()
        self.is_comic = False

    def handle_starttag(self, tag, attrs):
        attrs = dict(attrs)
        # If found 'comic' div, then next img ha
        if tag == 'div' and 'id' in attrs:
            if attrs['id'] == 'comic':
                self.is_comic = True

        # Set self.url to comic image url
        elif tag == 'img' and self.is_comic:
            self.url = attrs['src']
            self.is_comic = False


def get_xkcd(n=''):
    # Path to xkcd folder on Desktop
    folder = os.path.join(os.path.expanduser('~'

    # If folder doesn't exist, create one
    if not os.path.exists(folder):
        os.makedirs(folder)

    # Download comic page
    url = 'http://xkcd.com/{}'.format(n)
    page = urlopen(url).read().decode('utf-8')

    # Get image url from parser
    parser = Parser()
    parser.feed(page)

    image_name = parser.url.split('/')[-1]
    path = os.path.join(folder, image_name)

    with open(path, 'wb') as f:
        f.write(urlopen(parser.url).read())

# Get a bunch of comics
```

```
for i in range(1200, 1212):
    get_xkcd(i)
    print('Downloaded xkcd #{}'.format(i))
```

PowerShell:

```
# Function to download xkcd comic. Get latest or
Function GetXkcd($n='') {
    # Path to xkcd folder on Desktop.
    # [Environment] is a .NET class with static
    # Trailing backtick for line continuation.
    $folder = Join-Path -Path ([Environment]::Ge
                        -ChildPath 'xkcd'

    # If folder doesn't exist, create one
    if (-not (Test-Path -Path $folder)) {
        New-Item $folder -Type Directory
    }

    # Initiate web request (requires PowerShell
    # Uses -f to format string to replace {0} by
    $result = Invoke-WebRequest ("http://xkcd.cc

    # Get URL of image that is a comic
    $url = ($result.Images | where src -match /c

    # Join folder path with image file name
    $destination =  Join-Path -Path $folder `
                              -ChildPath ((Split

    # Download image
    $wc = New-Object System.Net.WebClient
    $wc.DownloadFile($url, $destination)
}

# Get a bunch of comics
foreach ($i in 1200..1211) {
    GetXkcd $i
    "Downloaded xkcd #{0}" -f $i
}
```

Notes

You might have to change your PowerShell execution policy to run scripts. To do this, run PowerShell as Administrator and type `Set-ExecutionPolicy RemoteSigned`.

A great way to learn about PowerShell is to use the `Get-Help` cmdlet. Simply type `Get-Help` followed by any other cmdlet to get more information about it. You can update your help files to be more comprehensive by running `Update-Help` as an administrator (PowerShell 3).

You can use IE to parse HTML instead of `Invoke-WebRequest` if you don't have PowerShell 3.

Conclusions

This is just scratching the surface of PowerShell by covering the syntax basics. The real power of PowerShell comes from piping and the various types of cmdlets that come built in to the language, similar to the Python standard library. You also get a nice IDE to boot - and it's already on your Windows machine.