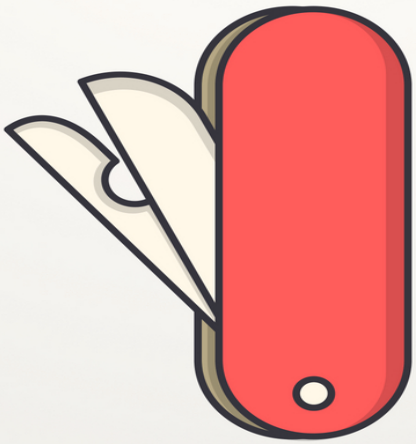
HOW TO UP YOUR SCRIPTING GAME ?

POWERSHELL GUIDE TO PYTHON

A COMPARATIVE APPROACH TO LEARN

TWO LANGUAGES AT ONCE. FAST!



PRATEEK SINGH

PowerShell Guide to Python

A Comparative Approach to Learn two Scripting Languages at once. Fast!

Prateek Singh

Tweet This Book!

Please help Prateek Singh by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought this amazing book: #PowerShell guide to #Python!

The suggested hashtag for this book is #PS2PY.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#PS2PY

Contents

Introduction

This PowerShell Scripting guide to Python is **designed to make readers familiar with syntax**, **semantics and core concepts of Python language**, in an approach that readers can totally relate with the concepts of PowerShell already in their arsenal, to **learn Python fast and effectively**, **such that it sticks with readers for longer period of time**.

"Use what you know to learn what you don't." also known as **Associative learning**.

PowerShell has been my scripting language of choice because of multiple reasons which includes its intuitiveness, command chaining with pipelines, smaller code and overall robustness! Undoubtedly it has shaped my career for good. As a scripting language it is very powerful when it comes to get the job done, although I do not intend to say that it is always the best tool to use. I feel when we love something we develop an deep affinity towards it, which seeds a bias towards it; and it grows into a preference and limits our learning somewhere. Which is just not right and I think we should be open to options and ideas, more specifically scripting/programming languages.

While PowerShell is still evolving in last 10 years since its inception, I see a lot of possibilities with more mature languages like Python.

Python is one of the top programming languages and in fast changing IT scenarios to DevOps and Cloud to the future - Data Science, Artificial Intelligence (AI) and Machine Learning Python is a must know. But this PowerShell Scripting guide to Python would be very helpful for you if you already have some knowledge of PowerShell.

Prateek Singh[1]

[1] https://twitter.com/SinghPrateik

About the Book

Why did I write this book

PowerShell is my go to scripting language for almost ANYTHING! but When I started learning Python, I realized that I could see many patterns and concepts that resembled with PowerShell to a great extent. Probably this is because PowerShell is a very new scripting language compared to Python, and has adopted best features from older scripting and programming languages.

Despite of the reasons, the reality is that both Python and PowerShell has lot of similarities and there is a huge deal of overlap in the core concepts. Thus during my journey to learn Python, I could very easily relate it to the concepts of PowerShell that I had cemented in my brain with years of usage and fiddling around. Being able to relate and associate the concepts from two languages, helped me to quick grasp Python in no time.

Science supports this and coins a word around this known as “Associative Learning”, that means you can “Use what you know (PowerShell) to learn what you don’t (Python)”. That is why the book follows a comparative approach to moon shoot the readers to their journey in Python, and it will help if you already know PowerShell.

Another aspect of learning Python could be if you are into System adminstration then, you start to see how everything that works in PowerShell from a perspective of a programming language. This will in return help you in understanding subtle concepts that seemed obvious in PowerShell, now at a deeper level.

Who is the Target Audience

Python is one of the top programming languages in world and in fast changing IT scenarios to DevOps and Cloud, to the future - Data Science, Artificial Intelligence (AI) and Machine Learning; Python is a must know.

- Any System Administrator who wants to step into Development or Programming roles, and even if you don’t want to be a developer, knowledge of another scripting language will make your skill set more robust.
- Python Developers who want to learn PowerShell scripting and understand its ease of user and importance to manage any platform.

Prerequisites

Some prior experience and knowledge of PowerShell is recommended, but even if you are not experienced in PowerShell this book gives an opportunity to learn PowerShell basics in an effective approach.

All scripts and code references in this book is either Windows PowerShell v5.1 or Python v3.6, and I'll highly recommend you to install them on your local machines, before proceeding with reading chapters and trying out code samples in this book. Following are links and installation steps:

- Windows PowerShell v5.1[2]
- Python v3.6 - I'm using Anaconda[3] that is bunch of useful Python packages that come pre-installed, and here[4] is a link to the Anaconda user manual if you want to download that.

Feedback

In case you have any questions, comments, or your helpful feedback about this book, please share it via email to prateek@ridicurious.com[5]

[2] https://docs.microsoft.com/en-us/powershell/scripting/setup/installing-windows-powershell?view=powershell-6
[3] https://www.anaconda.com/download/
[4] https://conda.io/docs/user-guide/install/download.html
[5] mailto:prateek@ridicurious.com

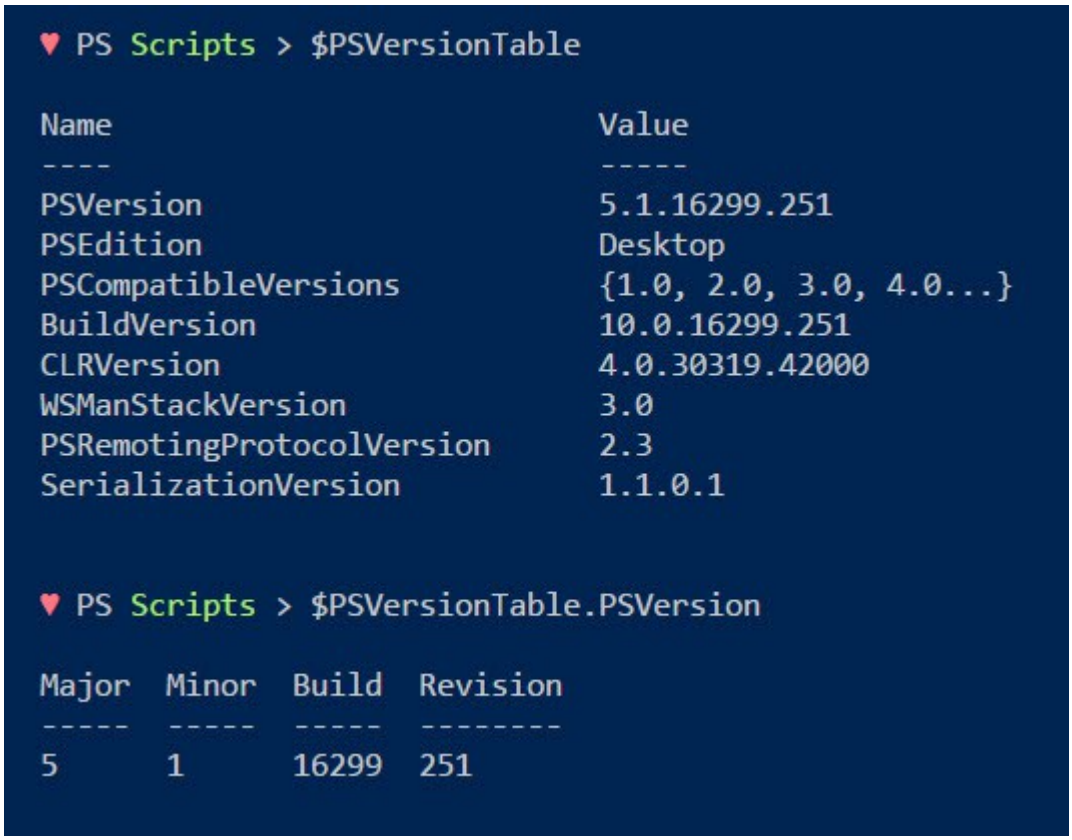
Chapter 1 - The Basics

This is the first chapter of the book and I want this chapter to make you more confident about your present skill. In this chapter We will discuss some basics of PowerShell and how that understanding could be used to learn Python, so that together we can jump start your Python Learning.

Version

The version is probably the first thing we learn to check in PowerShell, using the automatic variable `$PSVersionTable`

```
# checking version
$PSVersionTable
# to check major, minor, build and release version
$PSVersionTable.PSVersion
```

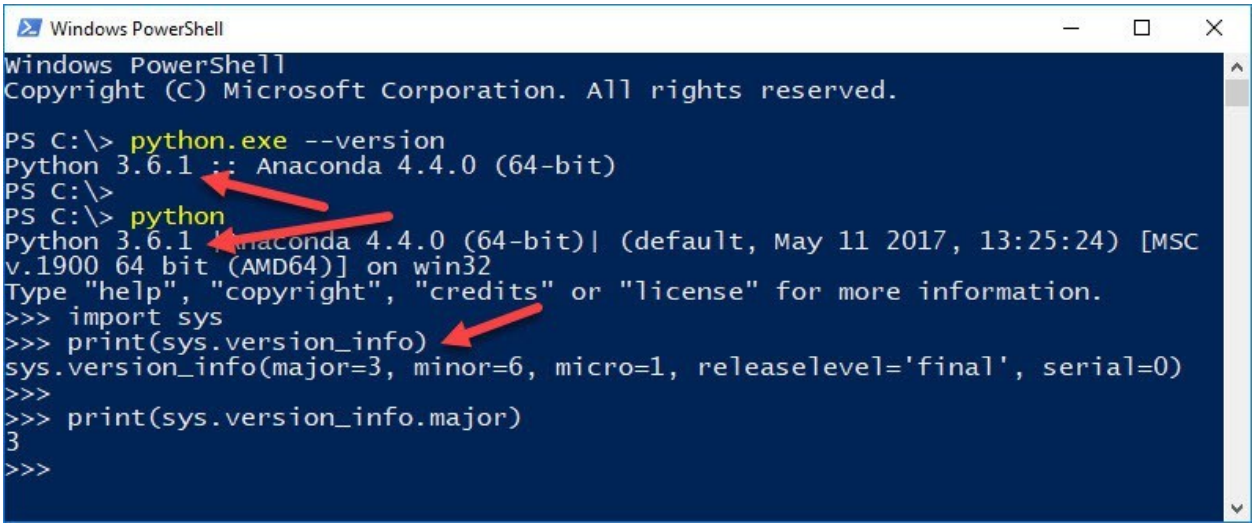


Version of PowerShell

While in Python the version automatically appears when you launch Python shell or call the python executable (`python.exe`) with the switch `--version`

There would be scenarios where you want to check the Python version dynamically during the code execution, for which you can `import sys` module and check the `version_info` attribute.

```
# checking version using 'python.exe' executable from Windows CMD prompt
python.exe --version

# checking version dynamically from within the code
import sys
print(sys.version_info)
```

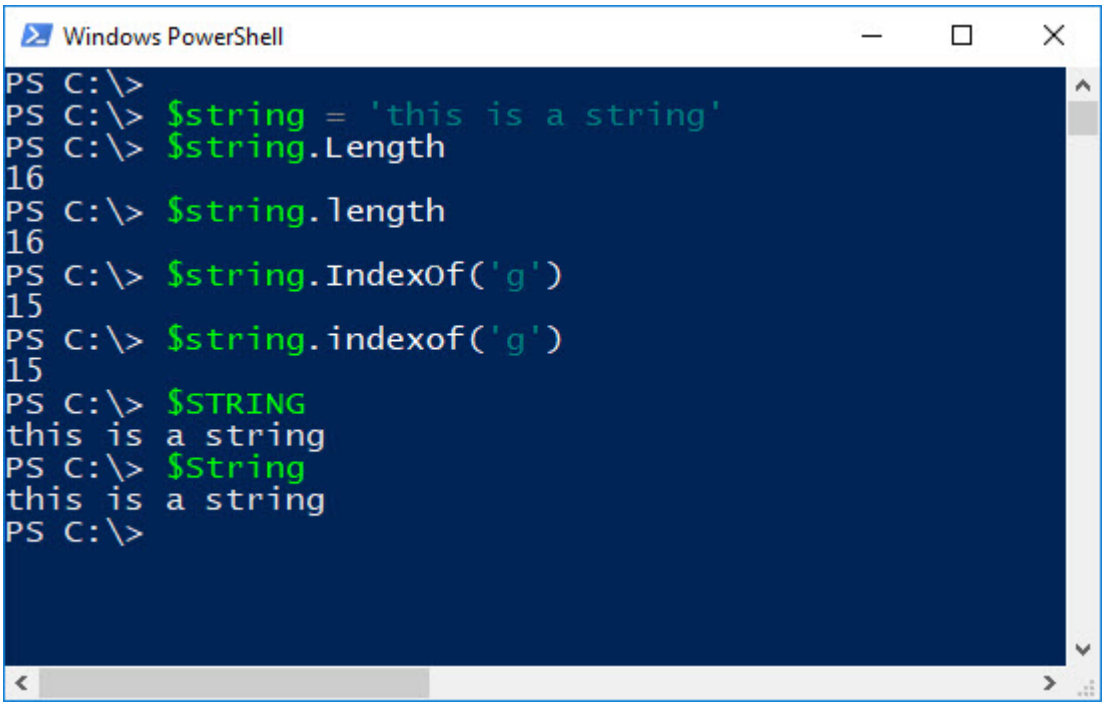


Version of Python

Case Sensitivity

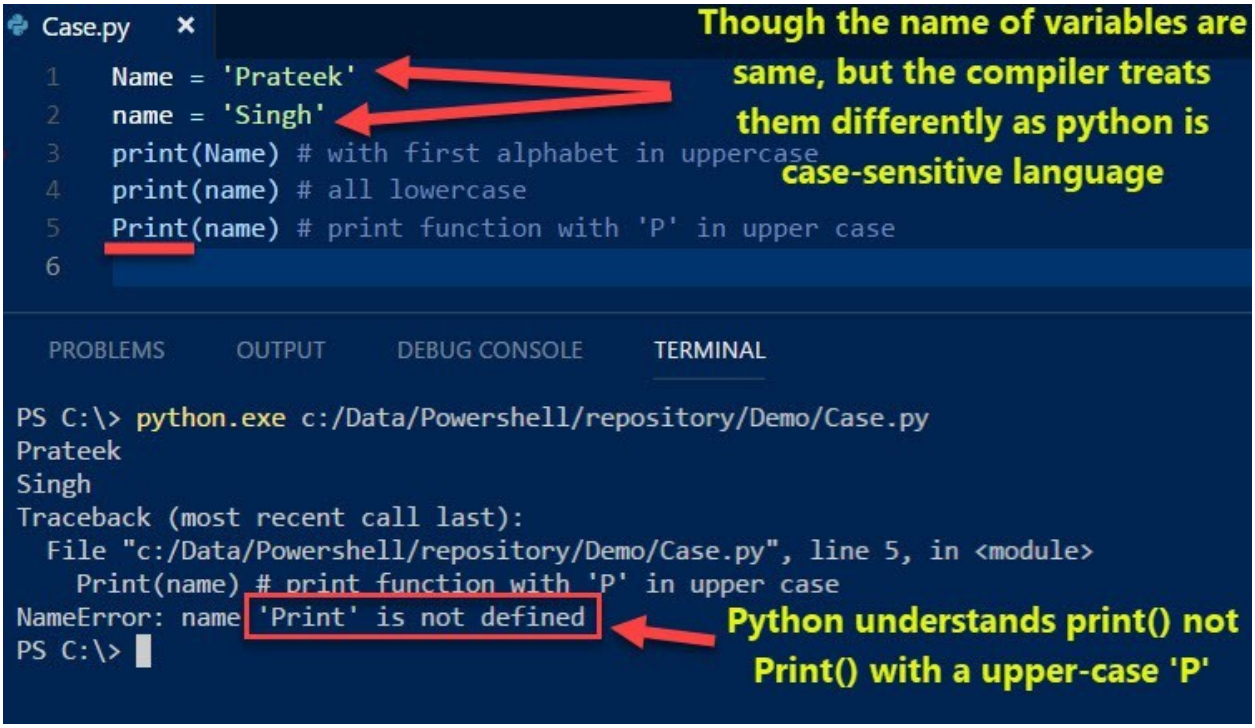
Powershell is a case-insensitive scripting language, that means in spite of the case of keywords, functions or the names of variables it is interpreted same.

```
# Changing of case of properties, methods and name of variable has no impact
PS C:\> $string = 'this is a string'
PS C:\> $string.Length
16
PS C:\> $string.length
16
PS C:\> $string.IndexOf('g')
15
PS C:\> $string.indexof('g')
15
PS C:\> $STRING
this is a string
PS C:\> $String
this is a string
PS C:\>
```



Case Sensitivity in PowerShell

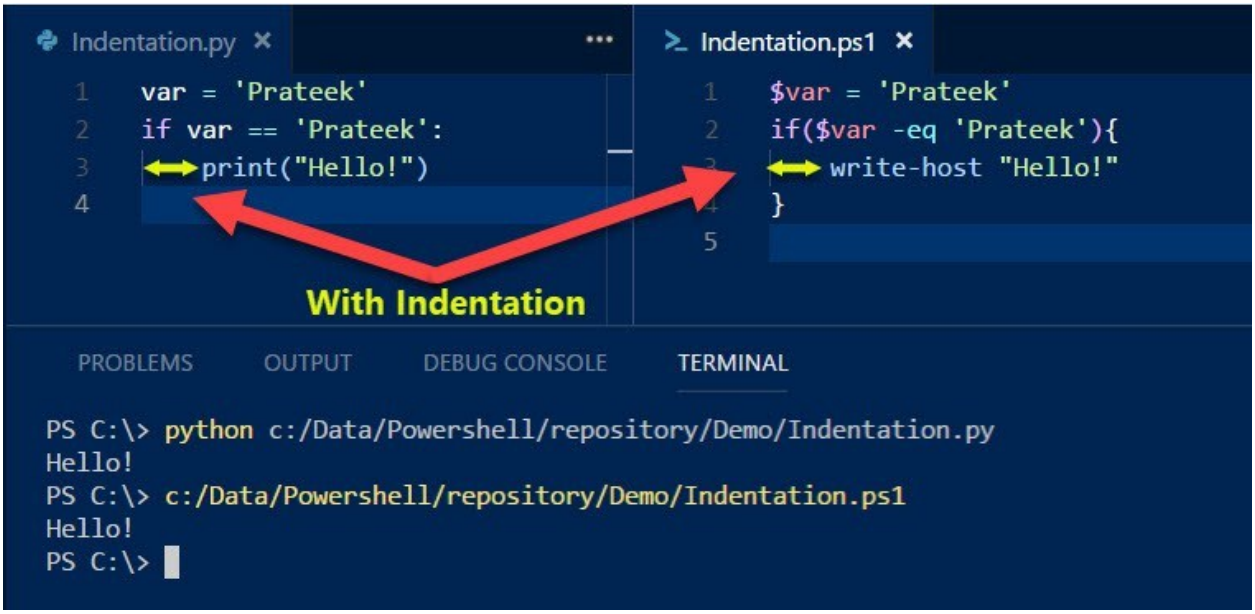
But **Python is strictly case-sensitive**, hence same keywords or names with a different case(s) are **understood differently by the compiler**.



Case Sensitivity in Python

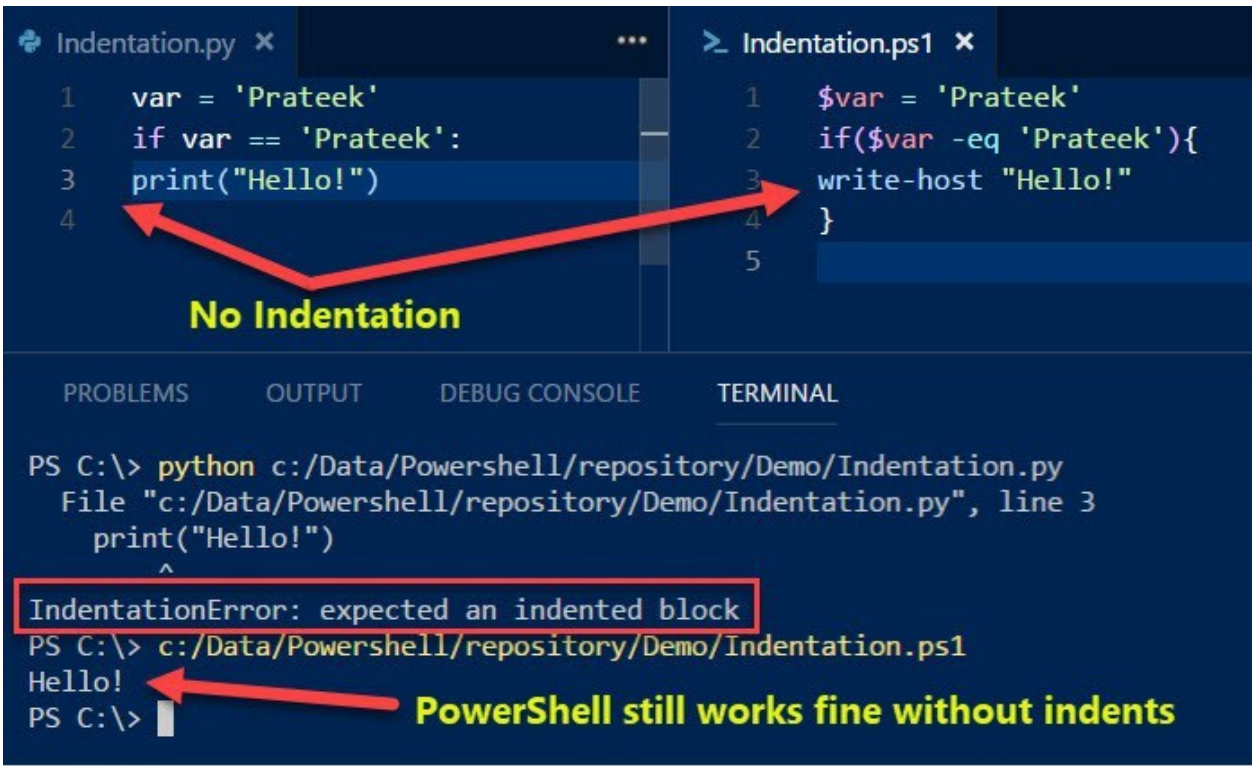
Indentation

Python is also strongly indented language. Indents provide structure to the program and differentiate code blocks from one another.



With the Indentation

In PowerShell Indentation is just a matter of style and is only used to make a script more readable and easy to understand, whereas **in Python Indentation is a language requirement**.



Without the Indentation

Comments

Single line comments Python is a **Hash sign (#)** followed by the comment.

```
# single-line comment in Python
```

which is exactly the familiar approach to make single line comments in PowerShell.

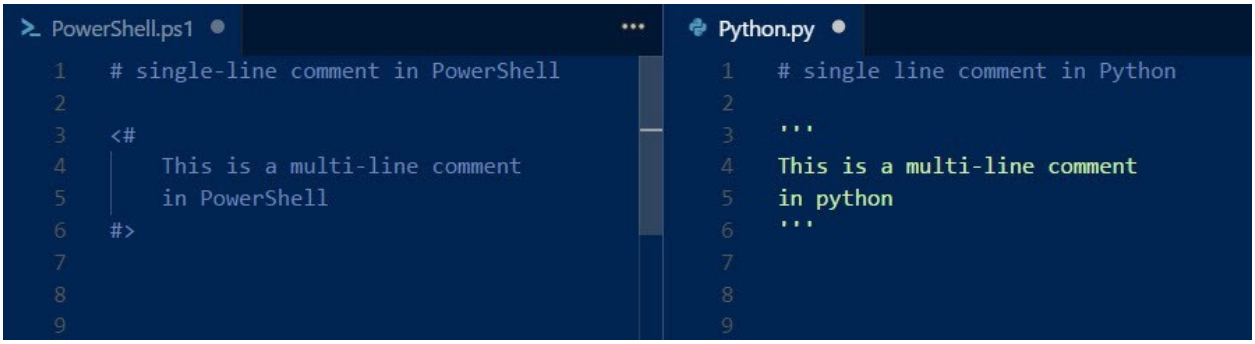
```
# single-line comment in PowerShell
```

On another hand, Python multi-line comments starts and ends with triple single or double quotes.

```
'''
this is a multi-line comment
in python
with single-quotes
'''

"""
this is a multi-line comment
in python
with double-quotes
"""
```

But in PowerShell, multi-line comments are formed using less-than (<) or greater-than (>) sign followed and preceded by a hash sign (#) respectively, as shown in the following code snippet.

```
<#
this is a multi-line comment
in python
with single-quotes
#>
```



Comments in PowerShell and Python

Variables

A variable is a unit of memory, typically used to store values like results from commands or expressions, paths, strings, settings etc.

There is no need to explicitly define Data types during the variable declaration in **Powershell and Python**, **as both are Dynamically typed programming languages**. That means data type of a variable is interpreted during the run time, depending upon the value the variable is holding.

Rules for naming PowerShell variables

Following rules must be followed when declaring a variable in PowerShell scripts:

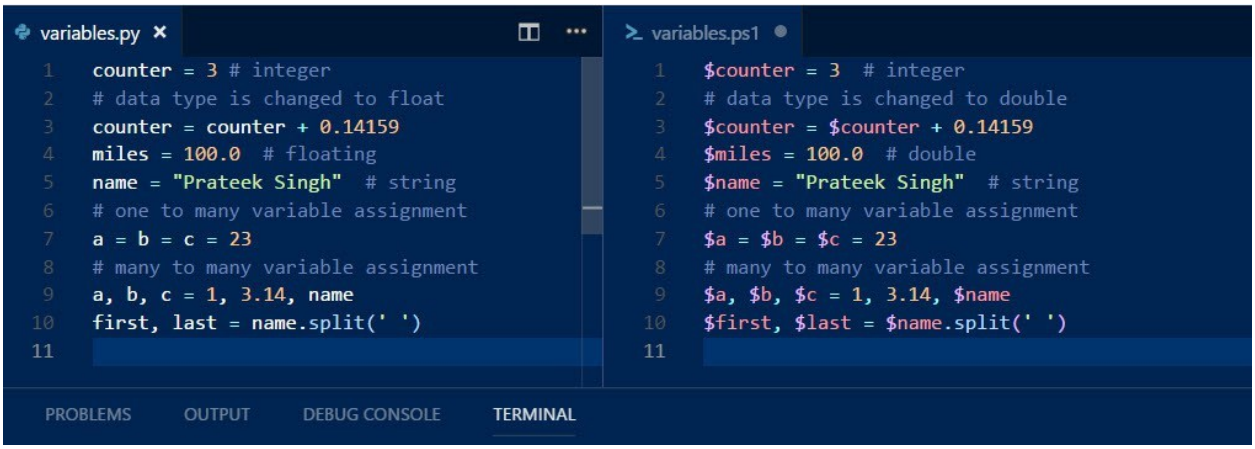
- PowerShell variables name are represented by text that **begins with a dollar sign ($)**.
- Variable names are **case-insensitive**, that means `$var`, `$Var` and `$VAR` represent the same unit of memory to store the data.
- Variable names **can include spaces and special characters**, but these are difficult to use and can be avoided because it is not very intuitive.

```
PS C:\> ${this is a variable} = 12
PS C:\> ${this is a variable}
12
```

Rules for naming Python variables

Following rules must be followed when declaring a variable in Python program:

- The first character **can be the underscore** "_", capital or lower-case letter.
- Following the first character can be anything which is permitted as a start character plus the digits.
- Since Python is **case-sensitive** so are the variable names, that means `var`, `Var` and `VAR` point to different units of memory.
- Python **keywords are not allowed** as variable names.



Defining Variables in Python and PowerShell

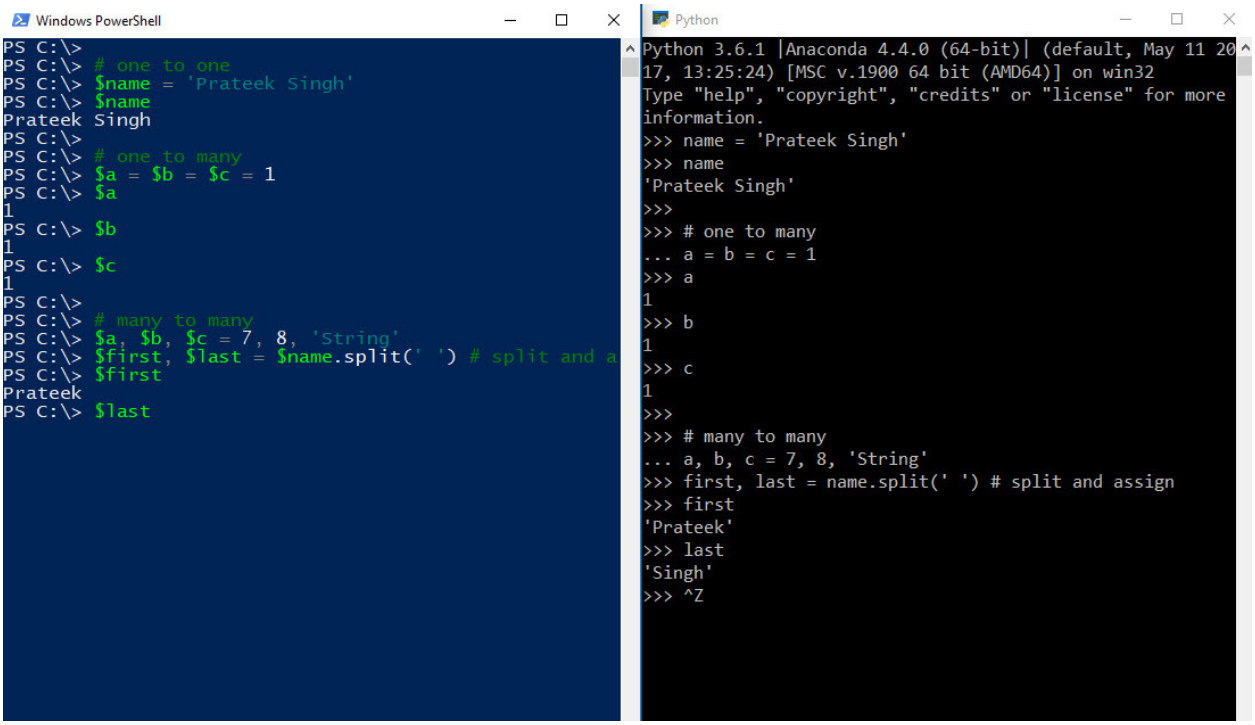
Multiple Variable Assignment

PowerShell allows you to assign multiple values to multiple variables at once. For example you can even perform a `.split()` method on a string and can store the parts of the string in different variables, as demonstrated in the following code sample:

```
# one to one
$name = 'Prateek Singh'

# one to many
$a = $b = $c = 1

# many to many
$a, $b, $c = 7, 8, 'String'
$first, $last = $name.split(' ') # split and assign
```

Python also allows exactly similar behavior of multiple variable assignments.

```
# one to one
name = 'Prateek Singh'

# one to many
a = b = c = 1

# many to many
a, b, c = 7, 8, 'String'
first, last = name.split(' ') # split and assign
```



Multi-Variable assignment

Key Pointers

- **PowerShell is a case-insensitive** scripting language, whereas are **Python is strictly case-sensitive**
- **Python is strongly indented** but, in **PowerShell Indentation is a matter of style**
- Both PowerShell and Python can have Single line comments using a **Hash or Pound sign (#)**
- All PowerShell variables **begins with a dollar sign ($)** and variable names are **case-sensitive**
- Python is **case-sensitive** language so are the variable names.
- Powershell and Python both are **Dynamically typed programming languages**, that means so you don't have to explicitly define the data type of the variable during declaration.

Reading Recommendations

To learn more, I'd advise you to please go through following PowerShell and Python help documentations from the console and/or find links to some additional external reading resources.

PowerShell Help

- Get-Help about_Variables

- Get-Help about_Automatic_Variables

Python Help

- Learn more switches that you can use with the Python executable: `python.exe --help`

Chapter 2 - Console Input/Output

Python and PowerShell both are interactive scripting languages which can capture the user data from standard input and display the results on standard output or on your console. In this chapter we are going to look into some of these approaches and best practices to read and print to PowerShell and Python consoles.

Printing to Console

Python provides a built-in function: `print()`, which by default prints the value passed to the standard output, or any other specified stream.

```
# console output
print("Hello World!")
# string with single quotes
print('Hello World in Single quotes')
# \ backslash is the escape char for the single quote
print('Hi, We\'re going to store')
print('Hello','world') # concatenated printing to console
print('Hi'+' There') # concatenated string with + plus sign
print('I am ',28,'years old') # concatenated Numbers and strings
print(19) # print number
```

Just like `Write-Host` or `Write-Output` cmdlet in PowerShell, in fact in PowerShell, it's much simpler and we can directly output strings to the console.

But there is a catch, `Write-Host` sends output directly to the console.

```
# console output
Write-host "Hello World!"

# to the output stream
Write-Output "Hello World!"

# Alternatively
"Hello World!"
```

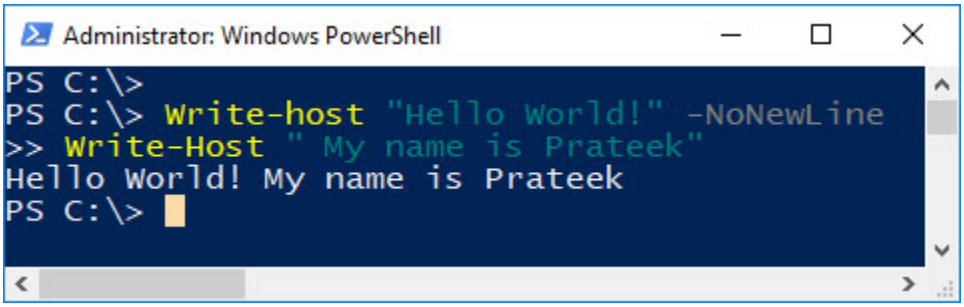
Strings inside quotes are sent to a different PowerShell Stream called Output stream (the default PowerShell stream), which can be piped and be used by cmdlets following the pipeline or can be stored in a variable.

Printing without a New-Line

Printing without a new line or in other words without adding a line feed character at the end of the output string is fairly simple in both PowerShell and Python.

In PowerShell when you don't want to add a line-feed you use `Write-Host` cmdlet with a `-NoNewLine` switch parameter, which will continue printing the next item just after the previous one, as demonstrated in the following example:

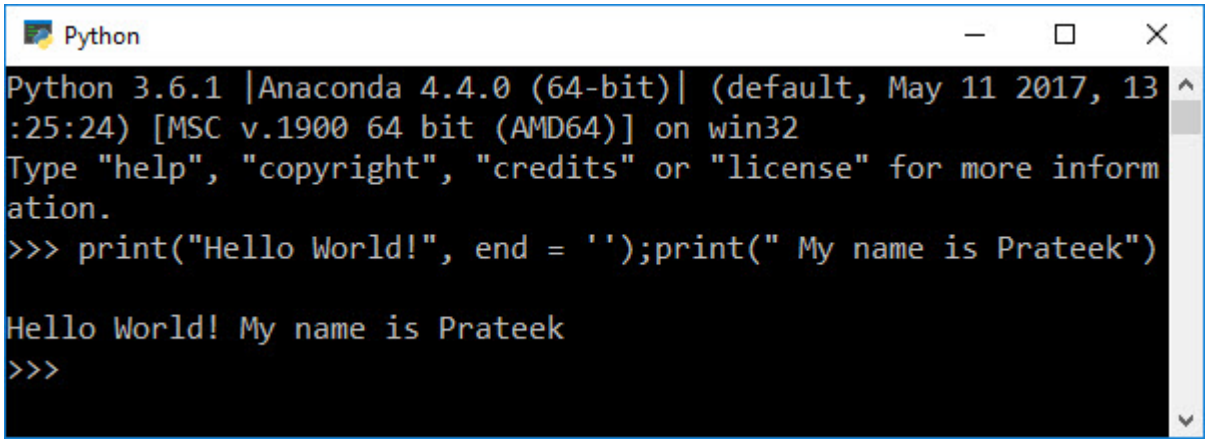
```
Write-host "Hello World!" -NoNewLine
Write-Host " My name is Prateek"
```



No New-Line in PowerShell

Whereas in Python, you've to use the named argument "end" to explicitly specify the end of line string, which should be an empty string (`''`) if you want to continue printing in the same line. The default value of the named parameter is `'\n'` which is a `new-line` character in Python and passing an empty string overwrites this value.

```
# use the named argument "end" to explicitly specify the end of line string
print("Hello World!", end = '')
print(" My name is Prateek")
```

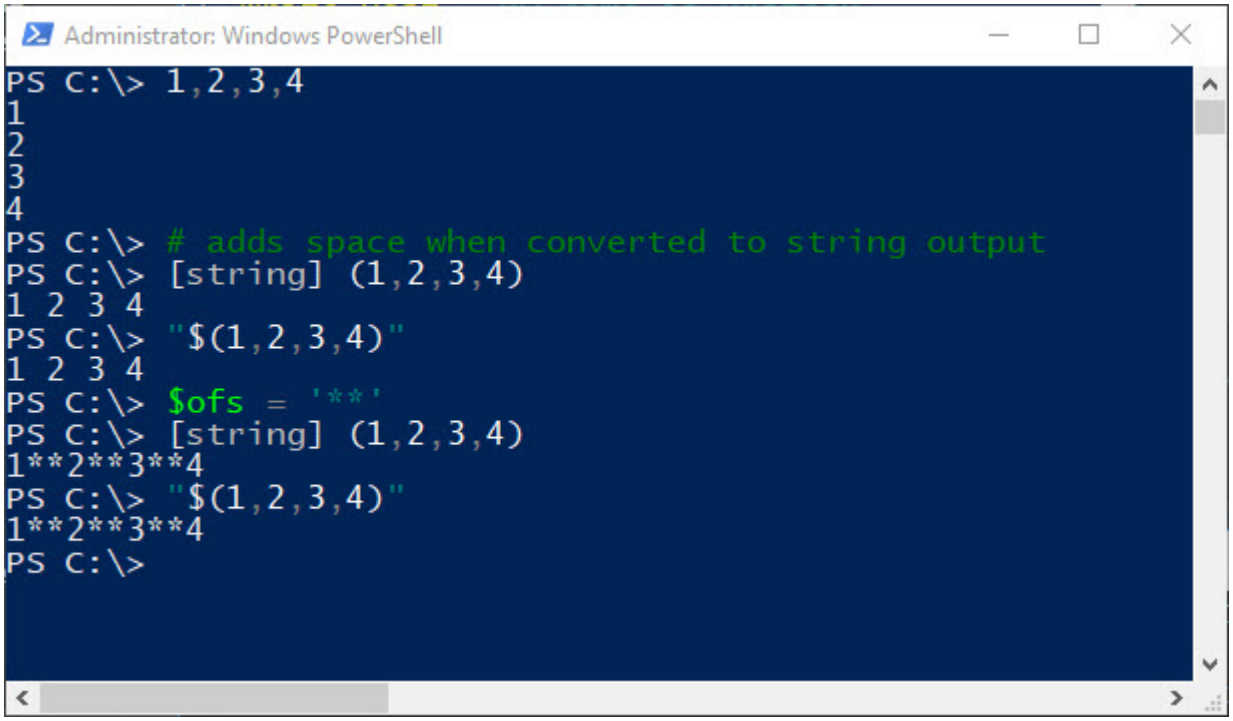


No New-Line in Python

Output Field Separator

PowerShell offers an Automatic Variable `$OFS` that can store a separator string, which is used as the 'Output Field Separator' between the objects printed on the console. The default value of this automatic variable is `" "`, that is why when you convert an array of `[int]` objects to string and print it to console it adds a space between each integer object in the output.

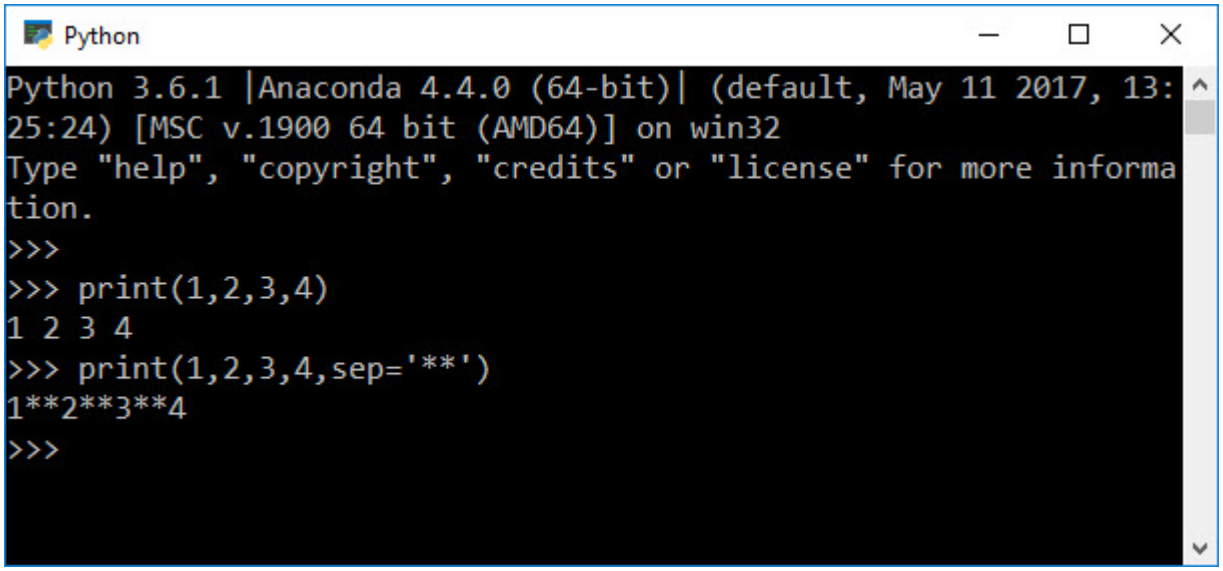
```
PS C:\> 1,2,3,4
1
2
3
4
PS C:\> # adds space when converted to string output
PS C:\> [string] (1,2,3,4)
1 2 3 4
PS C:\> "$(1,2,3,4)"
1 2 3 4
PS C:\> $ofs = '**'
PS C:\> [string] (1,2,3,4)
1**2**3**4
PS C:\> "$(1,2,3,4)"
1**2**3**4
PS C:\>
```



$OFS in PowerShell

`print()` method in Python also provides similar functionality to explicitly define the separator, using the named argument 'sep' which has a default value `sep=' '`. This is the reason for extra space introduced between integers printed using the `print()` function.

```
>>> print(1,2,3,4)
1 2 3 4
>>> print(1,2,3,4,sep='**')
1**2**3**4
>>>
```



'sep' attribute in Python's print() function

Console Output to a File

`print()` method can also redirect output to a file, but by default, it goes to standard output. There is a named argument '`file`' that allows you to specify a file name so that the output can be redirected.

```
# notice the forward slashes in the path
filename = open('c:/temp/file.txt', 'w')
print('Pretty cool!', file = filename)
filename.close()
```

On another hand, same can be achieved in PowerShell by piping the output string to `Out-File` cmdlet, which will then save the string in a file. A new file would be created if it does not exist.

```
# unlike in python the path is in backward slashes
"Way easier in PowerShell" | Out-File C:\temp\file1.txt
```

Output Streams can also be redirected to a file using '`>`' greater-than operator and '`*`' in the following command represent a wildcard for all the output streams.

```
# redirect the stream to a file
Write-Host "Using stream redirection" *> c:\temp\file2.txt
```

Append Console Output to a File

There would be some scenarios where you want to append the console output to a file and in order to achieve this in Python, you have to first open the file in append mode ('a') and then simply print the output to the file.

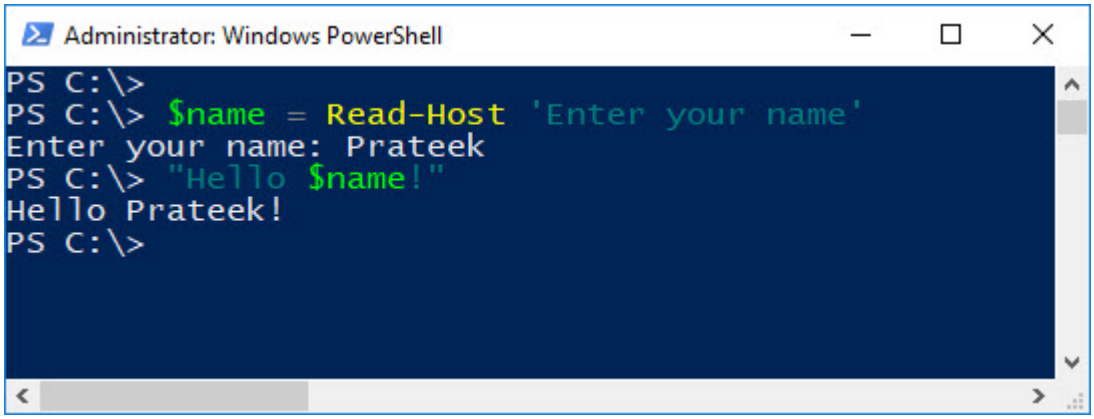
Whereas, in PowerShell you can use '>>' double `greater-than` operators to append the console output, while redirecting the streams. But with the `Out-File` cmdlet you should use the `-Append` switch parameter.

```
Write-Host "Using stream redirection" *>> c:\temp\file2.txt
"This is a string" *>> c:\temp\file2.txt
"Just add append switch" | Out-File c:\temp\file1.txt -Append
```

Reading from Console

Prompting input from users in the PowerShell console is as easy as using the cmdlet `Read-Host`, which can read a line of input from the console. You must specify a meaningful prompt string to the user for the input, and then capture user input to a variable.

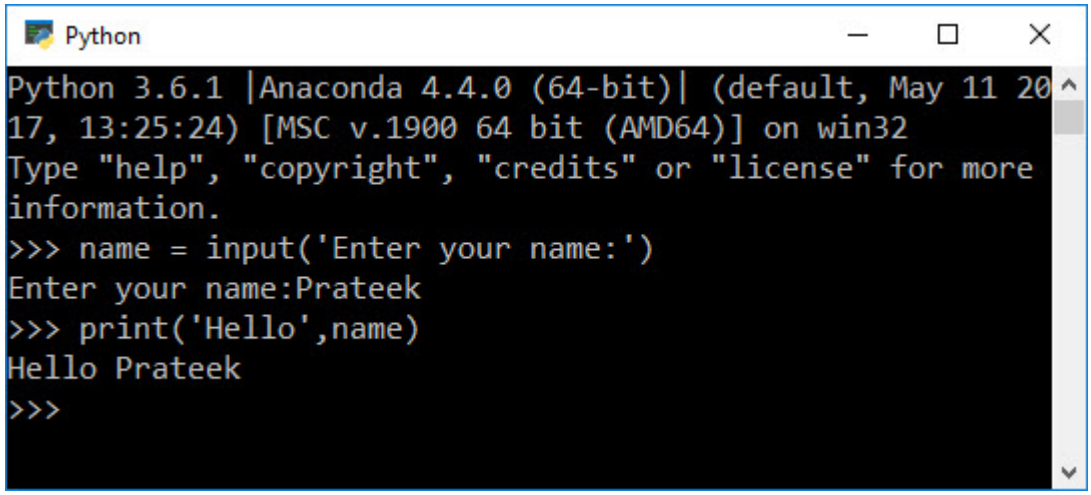
```
$name = Read-Host 'Enter your name'
"Hello $name!"
```



Reading from the host in PowerShell

Python on other hand provides a built-in method called `input()` to read a string from standard input from the console and the trailing new-line character (`Line-Feed`) when the user hits enter, is stripped away.

```
name = input('Enter your name:')
print('Hello',name)
```

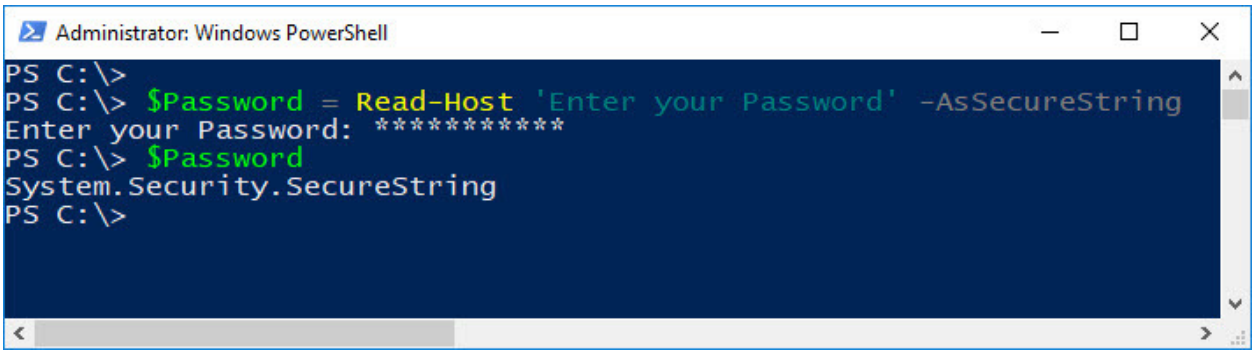


Reading from the host in Python

Reading from Console Securely

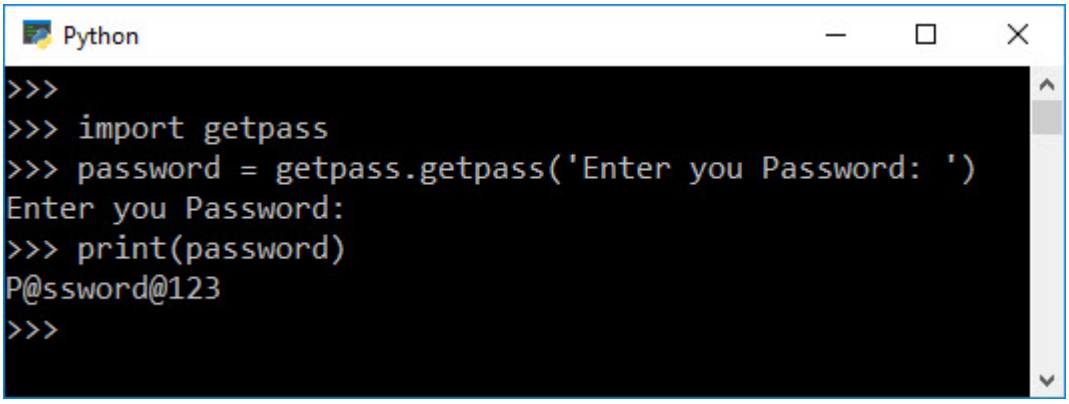
PowerShell's `Read-Host` cmdlet when used with `-AsSecureString` switch parameter will hide your input in case you prompt users for secure data, such as passwords.

```
$Password = Read-Host 'Enter your Password' -AsSecureString
$Password
```



Reading from host securely in PowerShell

While in Python you can use `getpass()` method to turn off echo while prompting users a secure data, but data captured can still be viewed in plain text. Before using the method you've to first import the `'getpass'` module into the current Python session.



Reading from host securely in Python

Key Pointers

* Python utilises `print()` function and PowerShell has `Write-Host` or `Write-Output` cmdlets to print output.

* Printing without a new-line

* PowerShell

```
1          Write-host "Hello World!" -NoNewLine
```

* Python

```
1          print("Hello World!", end = '')
```

- Built-in function `input()` is Python equivalent of PowerShell cmdlet: `Read-Host`
- `Read-Host` cmdlet in PowerShell with `-AsSecureString` switch parameter will hide user input to make it secure. Python has a `getpass()` Utility to get a password from a user with echo turned off.

Reading Recommendations

To learn more, I'd advise you to please go through following PowerShell and Python help documentations from the console and/or find links to some additional external reading resources.

PowerShell Help

- `Get-Help Write-Host`
- `Get-Help Write-Output`
- `Get-Help Read-Host`
- `Get-Help about_Redirection`

Python Help

- `help(print)`
- `help(input)`
- `import getpass; help(getpass)`
- `help(open)`

Links

- Understanding Streams and Redirection[a]

[a] https://blogs.technet.microsoft.com/heyscriptingguy/2014/03/30/understanding-streams-redirection-and-write-host-in-powershell/

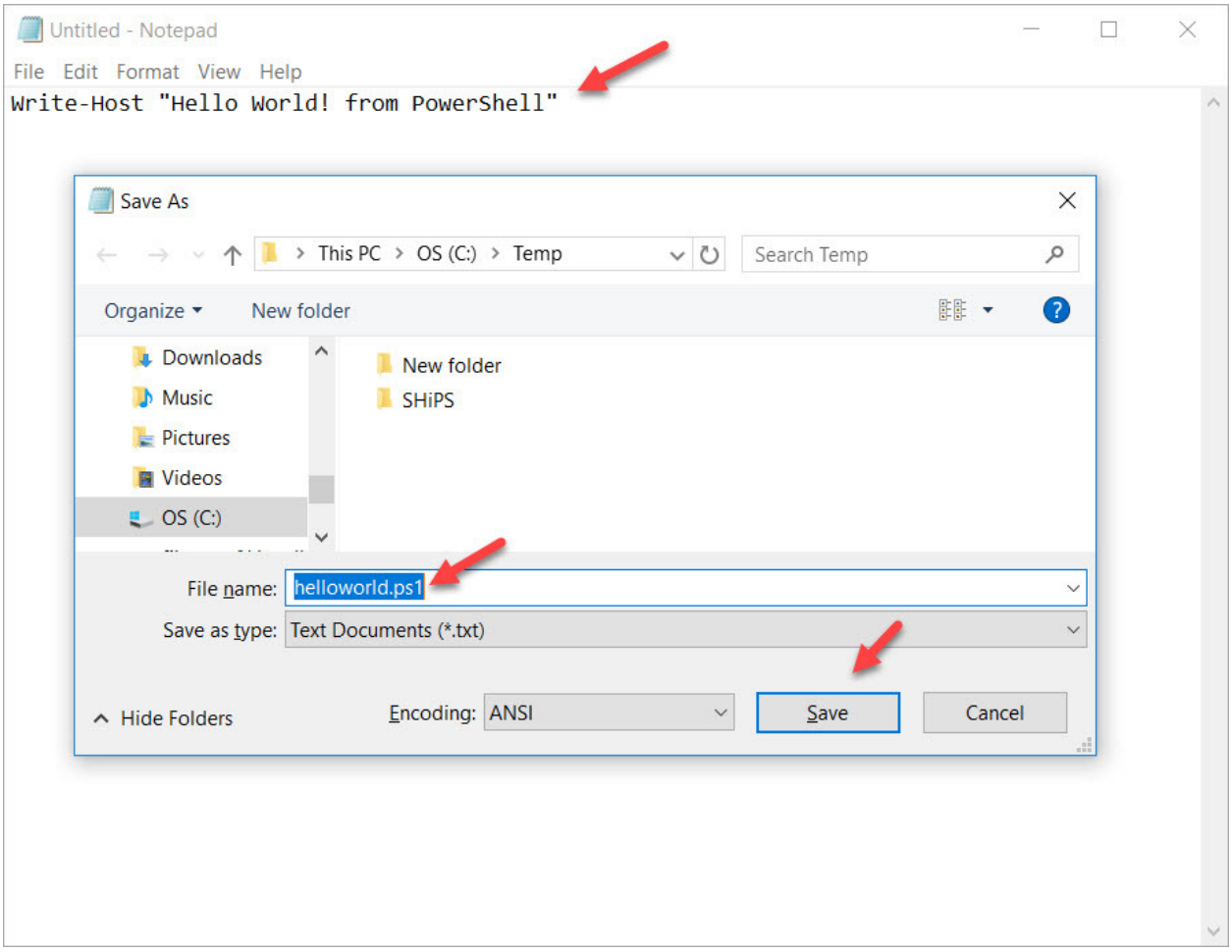
Chapter 3 - Creating your First Script

Scripts are programs written for a special runtime environment that can automate a list of tasks or operations, which would have been done otherwise manually by a person. A script can be anything between one to any number of lines of code, written to meet a certain requirement. So in this chapter, we will learn to create our first script and various approaches to run them in PowerShell and Python.

Saving Commands as a Script File

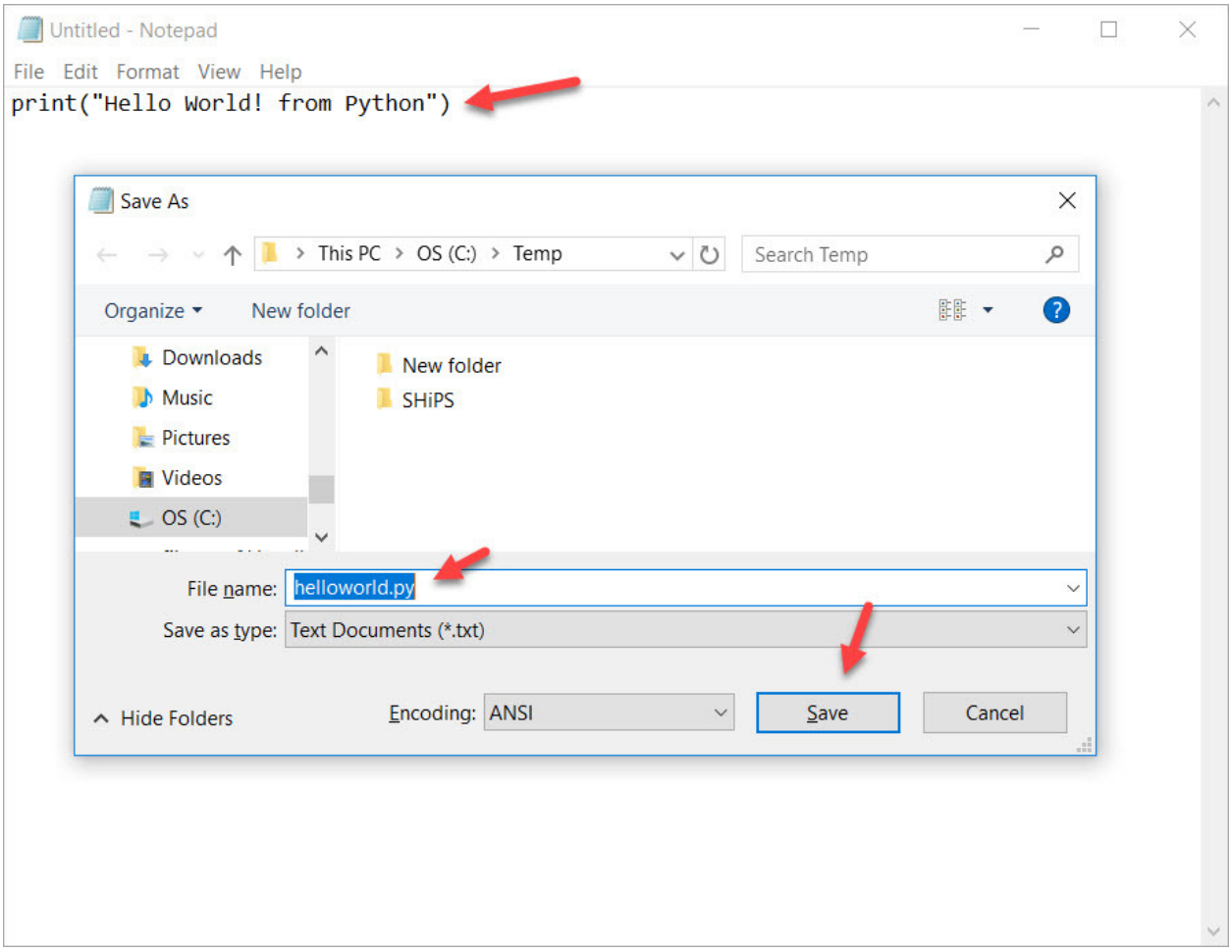
PowerShell scripts are just a text file saved with a `.ps1` extension and that is all you need to create a script. Simple!

Lets, take the easiest example of scripts that prints a string on the console.



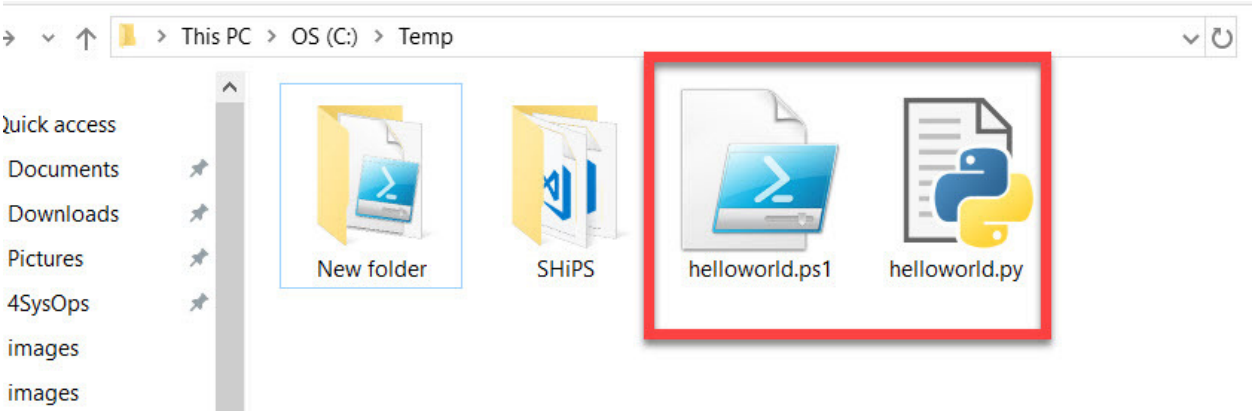
Save PowerShell scripts with .ps1 extensions

On other hand, Python scripts are saved as `.py` extension.



Save Python scripts with .ps1 extensions

Once the files are saved you can see them in File Explorer, with their respective icons as highlighted out in the following screenshot.



Saved Files

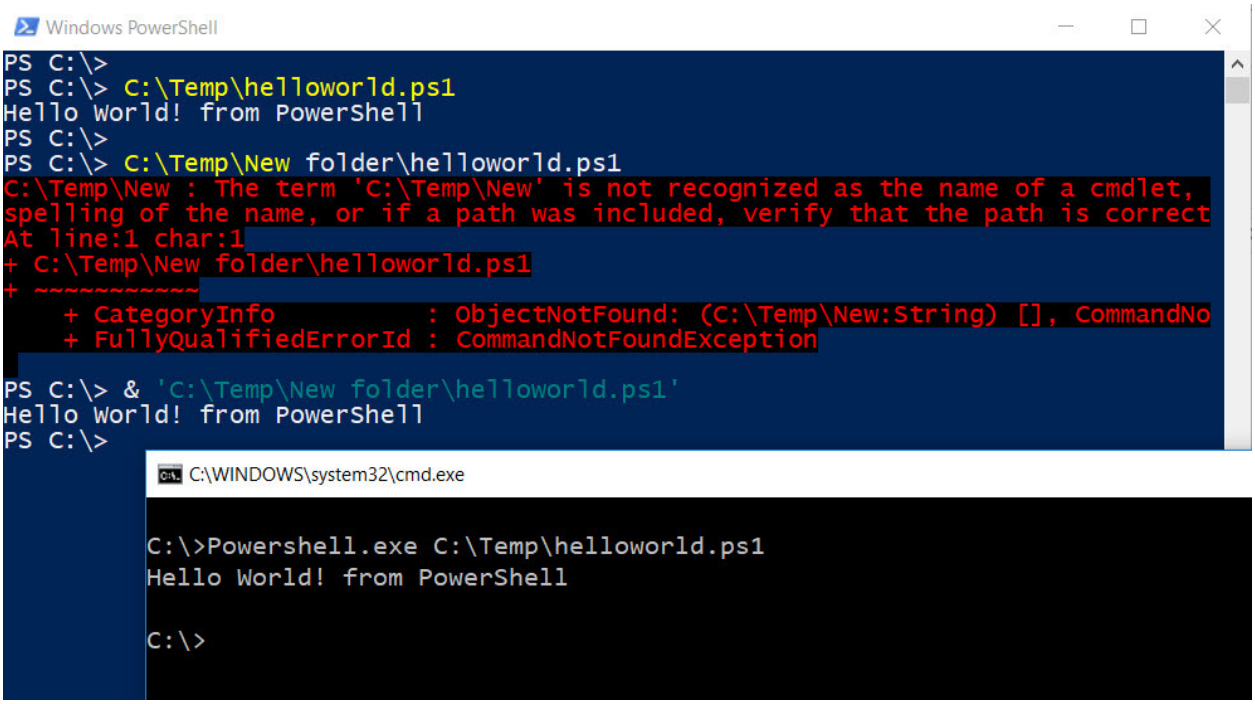
Running the Script File

There can be multiple ways to run a script file, depending upon what is the application that is running the script. Like PowerShell can run a script by just providing the full path of the script in the PowerShell console, in case the path has space, you can enclose the path in double-quotes (`" "`) or single-quotes(`' '`) to form a string and call the path string using the Call operator (`&`) also known as 'Ampersand', which will run your script.

```
# from PowerShell console type the full filename and hit enter
C:\Temp\helloworld.ps1

# in-case the file path contains space(s)
# use the '&' (Call Operator)
& 'C:\Temp\New folder\helloworld.ps1'
```

In some scenarios, you may want to run the script from the command prompt (CMD), and you can very easily call any PowerShell script by passing it as an argument to the PowerShell Executable (`PowerShell.exe`).

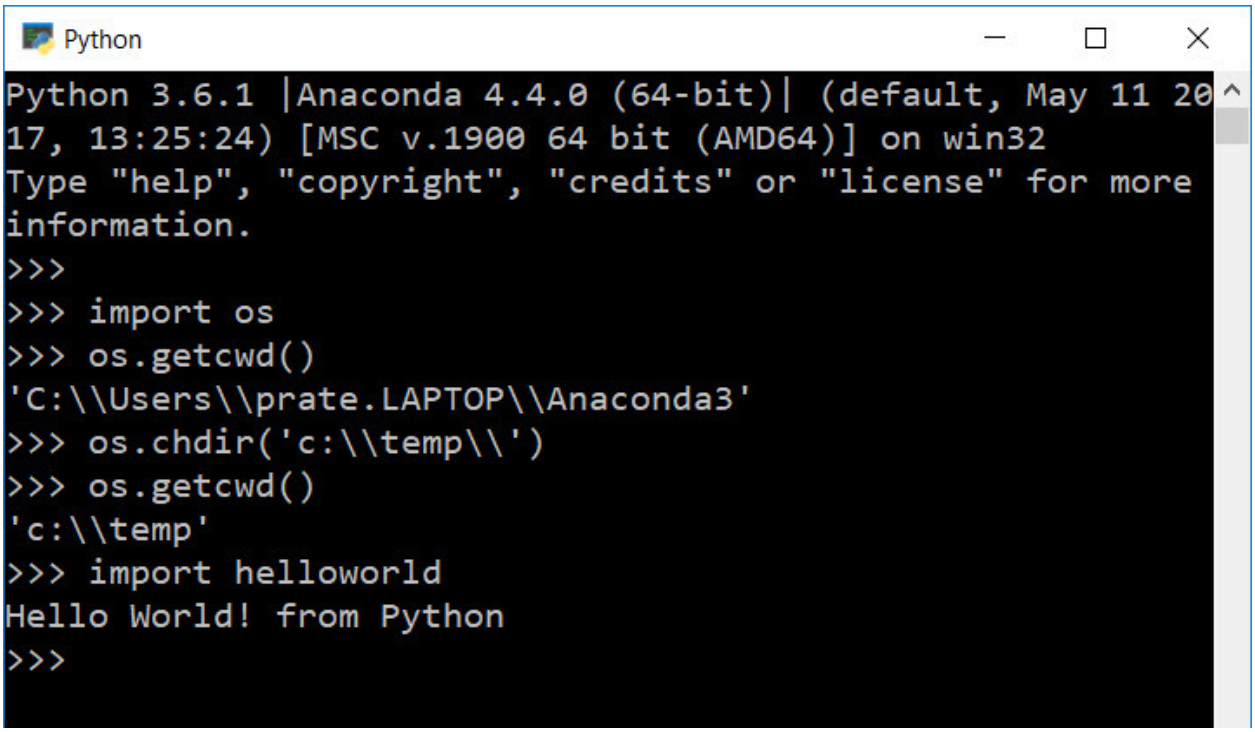
```
# or from the command prompt (CMD)
# using the PowerShell executable
Powershell.exe C:\Temp\helloworld.ps1
```



Running a PowerShell Script

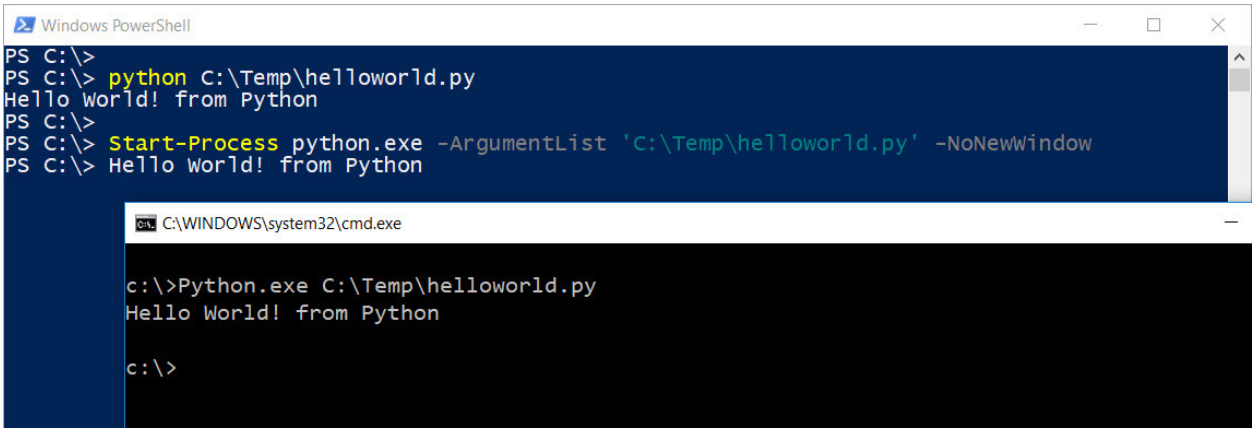
Similarly, Python scripts can also run from a Windows command prompt, Python Shell or even from the PowerShell console.

From the Python Shell, you've to first change the directory to the path which holds the script file using the `chdir()` method of the 'os' module, once you are in the directory import the file in the shell using the command `import <filename>` and it will run the Python script.



Running a Script from Python Shell

On another hand, in the command prompt and PowerShell, you can directly call the `Python.exe` and pass the script file path as an argument to the executable.



Running a Python Script

You can also use `Start-Process` cmdlet in PowerShell to run any Python scripts, by calling the Python executable and passing the script file path to `-ArgumentList` parameter as demonstrated in the following example:

```
Start-Process python.exe -ArgumentList 'C:\Temp\helloworld.py' -NoNewWindow
```

Key Pointers

- PowerShell scripts have a `.ps1` extension, whereas the name of the Python program file must end with `.py`.
- PowerShell programs can be called using one of the following methods:

```
# from PowerShell console
C:\Temp\helloworld.ps1

# using call operator
& 'C:\Temp\New folder\helloworld.ps1'

# PowerShell Executable
Powershell.exe C:\Temp\helloworld.ps1
```

- Run a Python program file by calling the Python executable (`Python.exe`) with the full file path.

```
# from the command line
python.exe C:\Temp\helloworld.py

# from the PowerShell console
Start-Process python.exe -ArgumentList 'C:\Temp\helloworld.py'
```

Reading Recommendations

To learn more, I'd advise you to please go through following PowerShell and Python help documentations from the console and/or find links to some additional external reading resources.

PowerShell Help

- Run `Get-Help about_Operators` from PowerShell console and go to the 'Call Operator (`&`)' section.
- `Get-Help Start-Process`
- `PowerShell.exe -Help`

- `Get-Help about_Redirection`

Chapter 4 - Passing Command-Line Arguments

Parameters are variables declared in the script and the arguments are the data you pass into these parameters. The whole purpose of passing arguments to a script is to change\tweak the behavior, output or functionality of script from outside the script without making any changes to the actual program. Both Python and PowerShell provide a couple of ways pass, access and parse arguments in the script, following are some such methods.

'sys.argv' in Python

Python has a 'sys' module or the `system module` that has a 'argv' attribute, which provides the list of command-line arguments passed to the Python program. It's basically an array holding the command line arguments of the program.

Index of this array starts at zero (0), not at one (1) and by default the first argument of the `argv` attribute i.e, `sys.argv[0]` is always the name of the program where it was invoked. So for example if we call a script file named: `FindMax.py` with arguments `13`, `23` and `57` then,

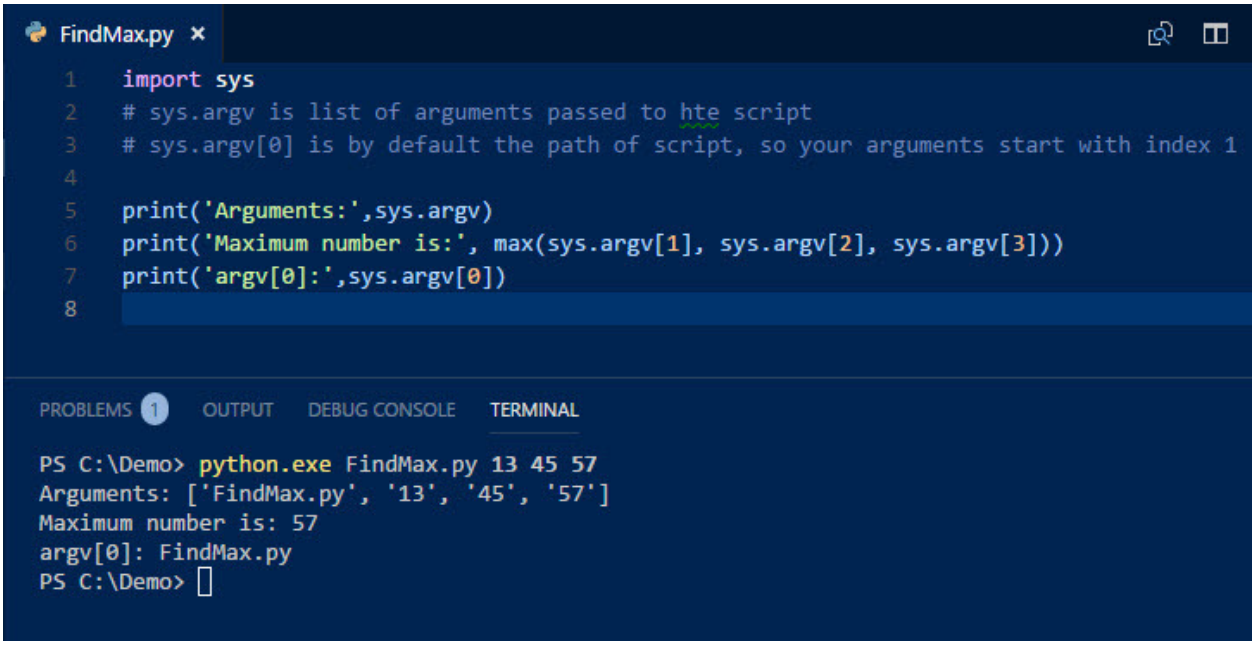
- sys.argv[0] is 'FindMax.py'
- sys.argv[1] is 13
- sys.argv[2] is 23
- sys.argv[3] is 57

Please save the following code snippet as a file named: "`FindMax.py`".

```
import sys
# sys.argv is a list of arguments passed to the script
# sys.argv[0] is by default the path of the script
# and user-defined arguments start with index 1

if len(sys.argv) > 1:
    print('Arguments: ',sys.argv)
    print('Maximum number is:', max(sys.argv[1], sys.argv[2], sys.argv[3]))
    print('argv[0]:',sys.argv[0])
```

And then pass the command line arguments to the script file: `FindMax.py` as demonstrated in the following example:

```
c:\> python.exe .\demo\FindMax.py 13 23 57

Arguments:  ['.\\demo\\FindMax.py', '13', '23', '57']
Maximum number is: 57
argv[0]: .\demo\FindMax.py
```



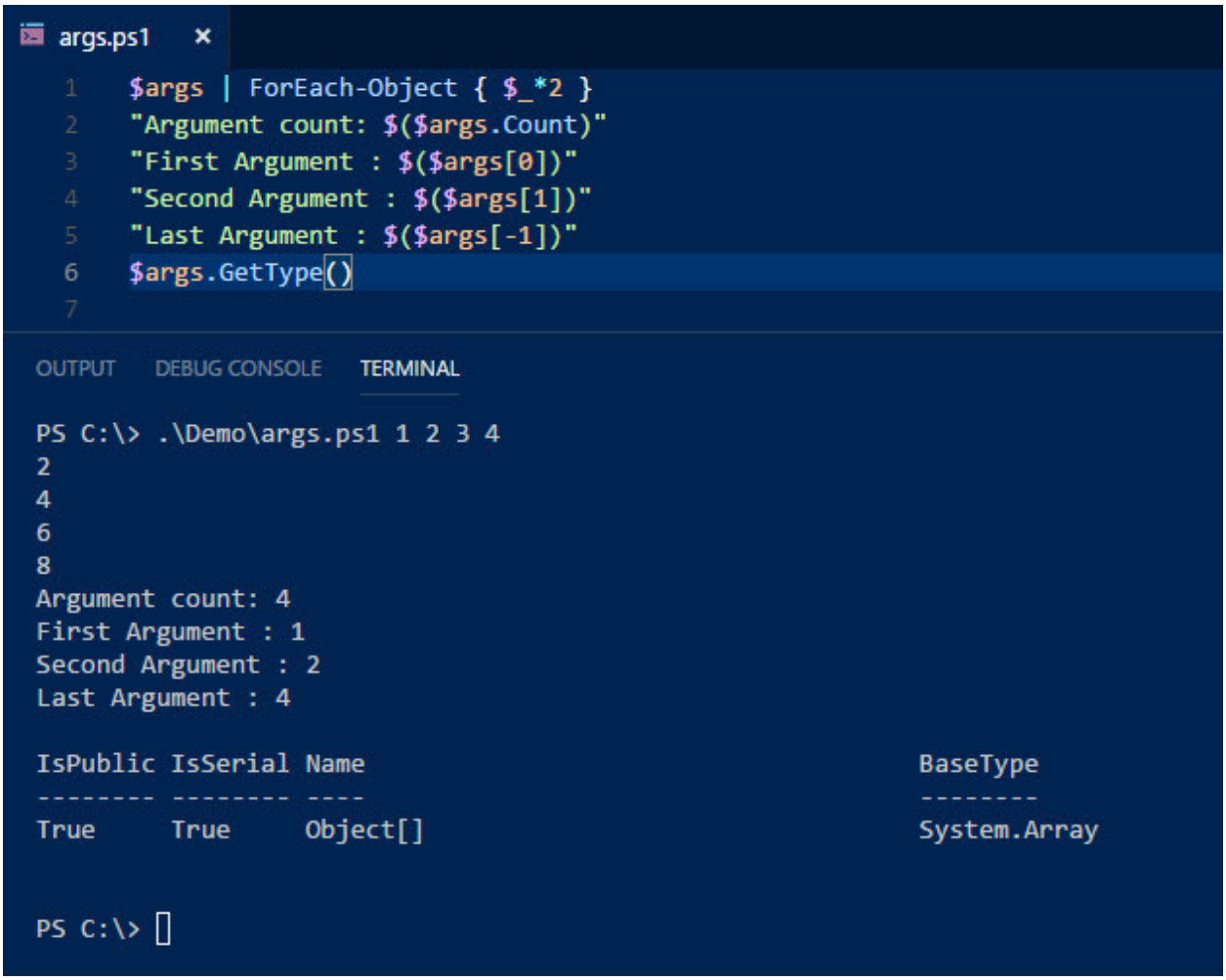
Passing Arguments to Python program

'$args' Automatic Variable in PowerShell

`$args` is an Automatic variable in PowerShell that contains an array of the undeclared parameters passed to a script, to know more check the help documentation in PowerShell using: `Get-Help about_Automatic_Variables`

```
# iterating through the arguments
# and performing an operation on each
$args | ForEach-Object { $_*2 }

"Argument count: $($args.Count)"
"First Argument : $($args[0])"
"Second Argument : $($args[1])"
"Last Argument : $($args[-1])"

$args.GetType()
```

Using the `$args` variable you can very easily perform basic operations on the arguments passed to the script, and since it is an array of arguments you can use the array indices to access the arguments. If there are more than one arguments, then `$args[0]` is the first argument and `$args[-1]` is the last argument passed to the script.



'Params()' in PowerShell

PowerShell allows users to create parameterized scripts with use of `param()` statement, which is simply a list of comma separated variables prefixed by data types like `[int]`, `[string]`, `[bool]` etc, these variables act as a parameter names.

If you are using `Param` statement in your script, then it should be the first thing (first-line) at the top of every command that executes in a script.

```
param (
    [string] $ComputerName = $env:ComputerName,
    [int] $NumOfCPU,
    [string]$UserName = $(throw "Please provide a UserName"),
    [bool] $Is64Bit = $false
    [switch] $Windows
)

@"
Name = $ComputerName
CPUCount = $NumOfCPU
UserName = $UserName
x64 = $Is64Bit
Windows = $Windows
"@
```

You can also define default values to these parameters using the assignment operator (=) like in the following example which will set the default value of the parameter `num` to `5` in case no values is passed as an argument.

```
param([int] $num = 5)
```

To call such parameterized scripts in PowerShell, you have to provide the path of the script file, then after a space use name of parameter prefixed with a hyphen (`-`) like `-UserName <value>` to pass arguments to the script parameter.

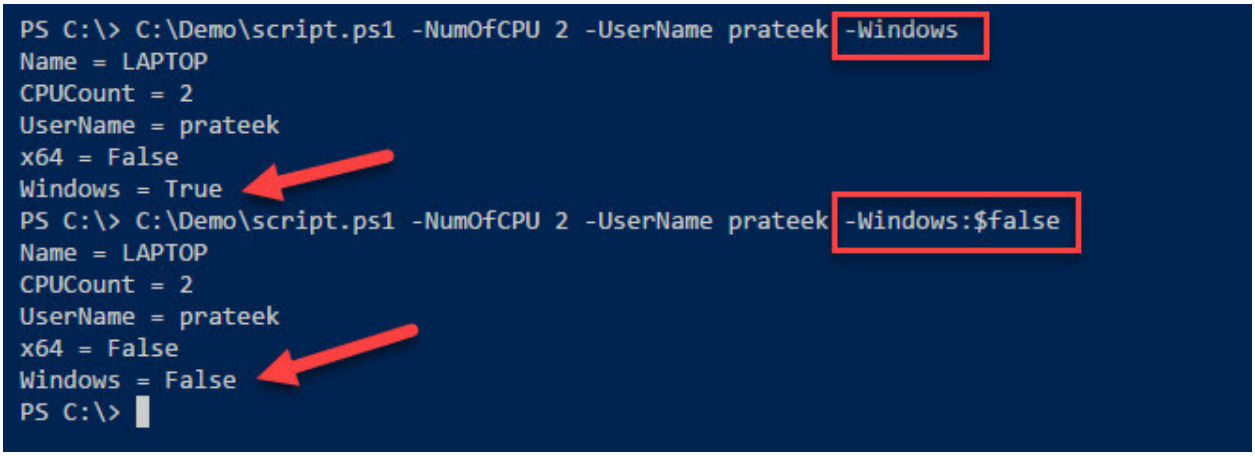
```
PS C:\> C:\Demo\script.ps1 -NumOfCPU 2 -UserName prateek
Name = LAPTOP
CPUCount = 2
UserName = prateek
x64 = False
Windows = False
```

In case you are using switch parameters like '`[switch] $Windows`' in our example, you can set the switch parameter to `$true` by specifying '`-Windows`'.

```
C:\Demo\script.ps1 -NumOfCPU 2 -UserName prateek -Windows
```

And to set switch parameters to `$false` use a colon ('`:`') just after the parameter name provide boolean FALSE (`$false`)

```
C:\Demo\script.ps1 -NumOfCPU 2 -UserName prateek -Windows:$false
```



Switch Parameters to a PowerShell Script

'argparse' Module in Python

The `argparse` module in Python is used to make user-friendly, powerful command-line interfaces. Depending upon the arguments defined in the program, `argparse` has the ability to parse the arguments passed to the script from the `sys.argv` attribute in the back-end.

The `argparse` module can also generate help and usage messages automatically, and throw errors in case program were passed invalid arguments.

If you want to know more about `argeparse` module, then please go through the Python's help system for full detailed help information

```
import argparse
help(argparse)
```

Creating the Argument Parser

Before you can begin defining parameters and parsing their arguments in your Python program, you've to first import the `argparse` module and create an object of class `ArgumentParser`, then store this parser object in a variable, like in the following example. Please note that the Name of a class in Python is always in `title-case` like `'ArgumentParser'`, not in lower-case.

```
# adding arguments
import argparse
parser = argeparse.ArgumentParser()
```

Adding the arguments

Once the argument parser object has been created, then we can add parameters and the arguments to the parser using `add_argument()` method of the `ArgumentParser` object. Let us take a simple example, and assume that you want to define a parameter named 'a' that should accept arguments of type integer (`int`) and you also want to provide a help message to this parameter. In order to implement this in Python, you have to follow the approach used in the following code sample.

```
parser.add_argument("a", help ="help text",type=int)
```

Here, `'a'` is the name of the parameter, '`type=int`' suggests that it accepts '`integer`' values and '`help ="help text"`' will add a help message to the parameter. This means the first argument passed to the program will be parsed and assigned to '`a`'.

Adding Multiple Arguments

In many scenarios you may want to specify more than one parameter and arguments, to do that just add more `parser.add_argument()` instances to define a sequence of arguments that will be parsed, like in the following example-

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("a", help="Provide a integer value", type=int)
parser.add_argument("b", help="Provide a integer value", type=int)
```

All the arguments defined above are mandatory and positional and the program will parse the arguments in the sequence they are defined. Something similar is also possible in PowerShell using '`param()`' statement.

```
param(
    [parameter(Mandatory, HelpMessage= 'Provide a integer value')] [int] $a,
    [parameter(Mandatory, HelpMessage= 'Provide a integer value')] [int] $b
)
```

Optional Arguments, Short Options, and Parameter Aliases

The Python's `argparse` module also allows the developers to define optional parameters by prefixing double-hyphen ('`--`') in the name of the parameter like '`--sum`' As the name suggests the optional parameter are not mandatory and you can run the script without passing arguments to such parameters, without any errors or issues.

The parameters defined using the `argparse` module can also have short options or one character aliases (optional parameters that are one character long) which are shorter to type, like '`-s`' and could be used instead of typing the complete name of optional parameters.

```
parser.add_argument("-s","--sum", help="Add the integers", action="store_true")
```

In the above example ' `action="store_true"` ' tells the parser that if the '`--sum`' or '`-s`' parameter is specified at the command-line, then the argument would be passed/stored as boolean `True`. If this optional parameter is omitted then it will the default value boolean `False` is assigned. Please refer to the following example and the screenshot to understand it better. This is the exact behavior observed when we define a `'Switch Parameter'` in PowerShell, like `[Switch] $Sum`

```
import argparse
parser = argparse.ArgumentParser()

parser.add_argument("a", help="Provide a integer value", type=int)
parser.add_argument("b", help="Provide a integer value", type=int)

# `True` is passed as an argument if the parameter is specified
parser.add_argument("-s","--sum", help="Add the integers", action = "store_true")

# parsing the cmdline-arguments
args = parser.parse_args()
print(args)
```

cmdline_args.py

```
import argparse
parser = argparse.ArgumentParser()

parser.add_argument("a", help="Provide a integer value", type=int)
parser.add_argument("b", help="Provide a integer value", type=int)
parser.add_argument("-s","--sum", help="Add the integers", action = "store_true")
args = parser.parse_args()
print("args:",args)
```

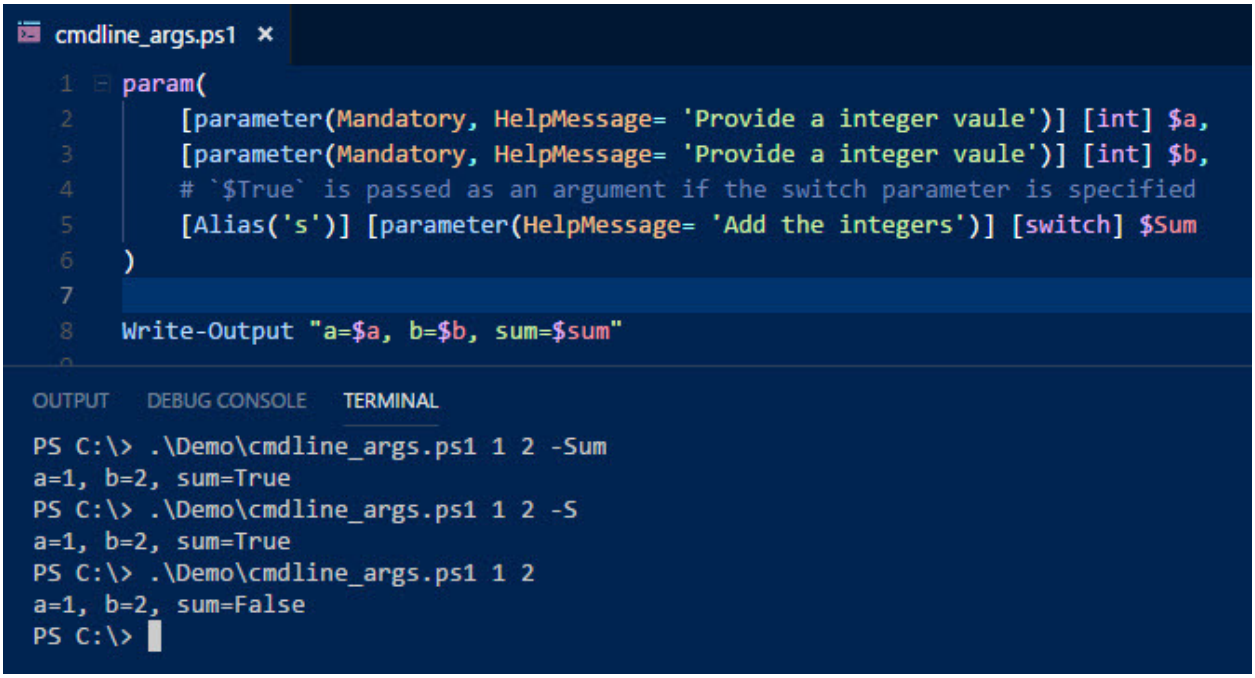
```
OUTPUT  DEBUG CONSOLE  TERMINAL

PS C:\> python.exe .\Demo\cmdline_args.py 1 2 --sum
args: Namespace(a=1, b=2, sum=True)
PS C:\> python.exe .\Demo\cmdline_args.py 1 2 --s
args: Namespace(a=1, b=2, sum=True)
PS C:\> python.exe .\Demo\cmdline_args.py 1 2
args: Namespace(a=1, b=2, sum=False)
PS C:\>
```

cmdline_args.py

PowerShell also has something equivalent for Short options, that are called the Parameter Aliases, which are utilized to define shorter names of the parameters, like in the following code sample using: `[Alias('s')]`.

```
param(
    [parameter(Mandatory, HelpMessage= 'Provide a integer value')] [int] $a,
    [parameter(Mandatory, HelpMessage= 'Provide a integer value')] [int] $b,
    # `$True` is passed as an argument if the switch parameter is specified
    [Alias('s')] [parameter(HelpMessage= 'Add the integers')] [switch] $Sum
)

Write-Output "a=$a, b=$b, sum=$sum"
```

Unlike parameters in Python's `argparse` module, in PowerShell by default any definition of a parameter is optional, and you've to explicitly add parameter attribute `[parameter(Mandatory)]` to make it mandatory.



Default Values

In PowerShell, we use assignment operator to assign a default argument value to a parameter, like `$n = 5` in the following example:

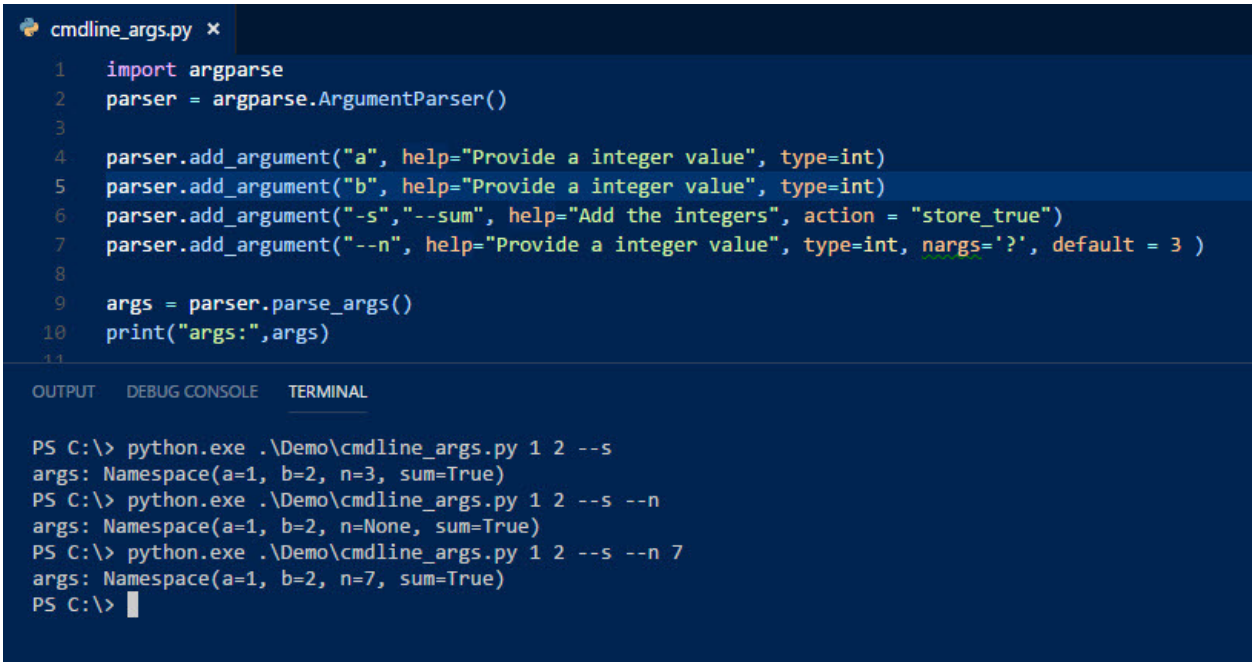
```
param(
    [parameter(Mandatory, HelpMessage= 'Provide a integer value')] [int] $a,
    [parameter(Mandatory, HelpMessage= 'Provide a integer value')] [int] $b,
    [Alias('s')] [parameter(HelpMessage= 'Add the integers')] [switch] $Sum,
    # Parameter with default value
    [parameter(HelpMessage= 'n-times')] [int] $n = 5
)

Write-Output "a=$a, b=$b, sum=$sum, n=$n"
```

Whereas, In Python, you've to pass the default argument to the named parameter `default` in the `parser.add_argument()` method, like in the below code sample as `default = 3`. This will pass the default argument value '3' to the parameter `--n` if it is un]]not explicitly specified in the command line arguments.

```
import argparse
parser = argparse.ArgumentParser()

parser.add_argument("a", help="Provide a integer value", type=int)
parser.add_argument("b", help="Provide a integer value", type=int)
parser.add_argument("-s","--sum", help="Add the integers", action = "store_true")

# Default argument will be passed to '--n' parameter
parser.add_argument("--n", help="Provide a integer value", type=int, nargs='?', defa\
ult = 3 )

args = parser.parse_args()
print("args:",args)
```

Here, `nargs` is the number of command-line arguments that should be consumed and `?` means, that number must be `0` or `1`. Defining the number of arguments (`nargs`) allows us to control how Python parameters will behave with/without argument and even when you don't specify them, like in following use cases.

```
PS C:\> # without parameter '--n' or arguments
PS C:\> python.exe .\Demo\cmdline_args.py 1 2 --s
args: Namespace(a=1, b=2, n=3, sum=True)

PS C:\> # with parameter '--n' but no arguments or 'None'
PS C:\> python.exe .\Demo\cmdline_args.py 1 2 --s --n
args: Namespace(a=1, b=2, n=None, sum=True)

PS C:\> # with parameter '--n' and with argument = 7
PS C:\> python.exe .\Demo\cmdline_args.py 1 2 --s --n 7
args: Namespace(a=1, b=2, n=7, sum=True)
PS C:\>
```



Parameter Help

To understand the mandatory/optional parameter set, the PowerShell script file can be passed to the `Get-Help <file-path>` cmdlet to return the parameter-set defined at the script level.

```
PS C:\> # Get-Help on Parameterized script files
PS C:\> Get-Help .\Demo\cmdline_args.ps1
cmdline_args.ps1 [-a] <int> [-b] <int> [-Sum] [<CommonParameters>]
```

Similarly, Python executable `Python.exe` has an optional parameter '`--help`' (also a short option '`-h`') that shows the help message with all the parameters defined in a Python program.

```
PS C:\> # Use the optional parameter '--h' to get parameters if Python program
PS C:\> Python .\Demo\cmdline_args.py --help
usage: cmdline_args.py [-h] [-s] [--n N] a b

positional arguments:
  a           Provide a integer value
  b           Provide a integer value

optional arguments:
  -h, --help  show this help message and exit
  -s, --sum   Add the integers
```

```
  --n N       Provide a integer value

PS C:\> # or the Short option '-h'
PS C:\> Python .\Demo\cmdline_args.py -h
usage: cmdline_args.py [-h] [-s] [--n N] a b

positional arguments:
  a           Provide a integer value
  b           Provide a integer value

optional arguments:
  -h, --help  show this help message and exit
  -s, --sum   Add the integers
  --n N       Provide a integer value
PS C:\>
```

Both Python and PowerShell parameters and/or arguments that are in square brackets ('`[]`') are optional and anything outside is mandatory. If you look closely in Python's help messages you'll notice that it also clearly labels the positional (mandatory) and the optional arguments separately, with a custom help message defined using the `argparse` module's `add_argument(help="Help Text")` method.

Key Pointers

- Python has `argv` dynamic object from the `sys` module that holds a list of command-line arguments, which is PowerShell's equivalent of `$args` Automatic variable.
- `sys.argv[0]` is always the name of the Python file to which command line arguments are passed.
- PowerShell `param()` statement allows users to create parameterized scripts, simply with a list of comma separated variables with data types like `[int]`, `[string]`, `[bool]` etc, that become the parameters for the script file.
- Python alternative to `param()` statement, is the `argparse` module that can make command line interfaces for a program\script and has abilities to parse the arguments passed.
- All the arguments defined using the `add_argument()` method of `'ArgumentParser'` class from the `argparse` module are by default mandatory and positional. The program will parse the arguments in the sequence they are defined.
- Unlike parameters in Python's `argparse` module, in PowerShell by default any definition of a parameter is optional, and you've to explicitly add parameter attribute `[parameter(Mandatory)]` to make them mandatory.
- Python and PowerShell parameters and/or arguments that are in `Parameter Help` with the square brackets ('`[]`') are optional, otherwise mandatory.

Reading Recommendations

To learn more, I'd advise you to please go through following PowerShell and Python help documentation from the console and/or find links to some additional external reading resources.

PowerShell Help

- `Get-Help about_Automatic_Variables` and look go through the `$args` section

Python Help

- `import sys; help(sys.argv)`
- `import argparse; help(argparse)`

Links

- Python v3 Argeparse Tutorial[a]

[a] https://docs.python.org/3/howto/argparse.html

Chapter 5 - Object Introspection

PowerShell and Python both are object-oriented programming languages, which means that almost everything is implemented using a special programming construct called classes, which are used to create objects that have properties and functionalities, like any real-world object. Before we dig deep into this, let's just quickly understand the basics of `classes`, `objects` and its members.

Class, Object, Property, and Method

Class

Generally speaking, a class is a category, set or a classification that has some attribute in common and can be differentiated from other classes by kind, type, or quality. For example, 'Human' is a class, and that is different from 'Animal' class. But in terms of programming languages, especially Python and PowerShell a `class` is a template for creating instances of the class, also known as objects.

The simplest way to create or define a class in Python and PowerShell is using the `class` keyword. For an example, let's create a simple class 'Human', with some properties and functionalities in Python, first thing first, you have to use the `class` keyword with the name of the class, followed by a colon (:). In the next line we change the indentation and provide a body of this class, as demonstrated in the following example:

```
1  class Human:
2      # this is a property/attribute
3      name = 'homo sapiens'
4      height = 5
5      # 'def' is used to define a method/function of a class
6      def eat(self):
7              print(self.name,'is eating now')
```

PowerShell on other hand has some small syntactical changes, like the `class` body is now enclosed in brackets `{ }` and we are not using `def` keyword to define methods like in Python. But overall there are a lot of similarities how a basic class is defined in both scripting languages, as you can deduce from the previous and the following example of a class.

```
class Human {
    # this is a property/attribute
    [String]$name = 'homo sapiens'
    [Int] $height = 5
    # define a method/function of a class
    eat(){
        Write-Host $this.name 'is eating now'
    }
}
```

Object

Objects are instances of a category (class) that has some attributes (properties) and performs some functions (methods). Just for the sake of making things simpler here let us suppose there is a category or a classification called as 'Human', Humans represent us together as a species, but if you take an instance of this class Human, let's suppose 'John' then John is an Object of the class human. 'John' has all attributes and functionalities of the class 'Human' of which he is an instance because all objects are created from the same class template.

Attributes or Properties

Attribute or Property is just another word for the characteristic of an object, like -

- name
- height
- weight

Functionalities or Methods

Functionalities are actions performed by the object and can be called Methods in terms of programming languages.

- eat()
- sleep()
- talk()

Note

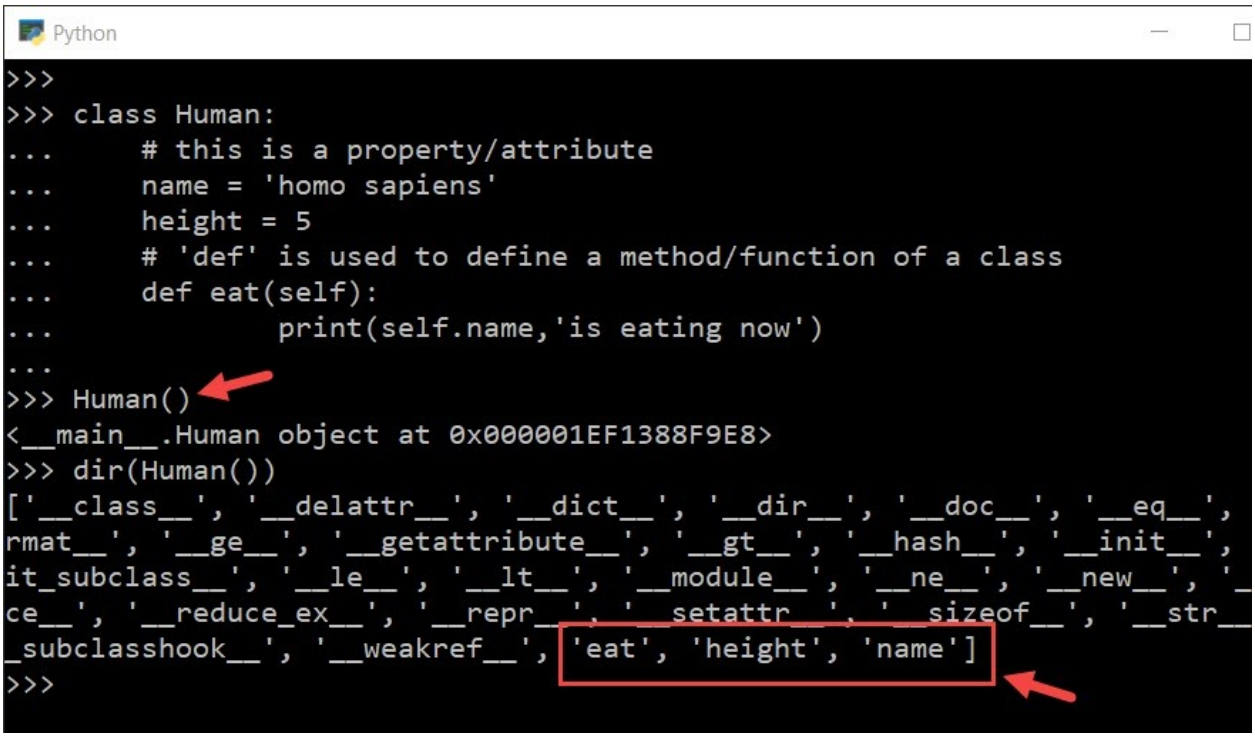
Functions and Methods are often used interchangeably, which is incorrect.

- A Method is a function defined inside the body of the class and to access a method you have to use the (.) Dot operator on the object of the class.
- A function on another hand can be a standalone, not necessarily in a class.

Instantiating the Class

To create a Python object or an instance of a class, you simply use the following syntax `<class name>()` and assign it to a variable for later use. It is a good practice to start the name of `class` with the first alphabet in upper case.

```
# instance of a class also called an Object
john = Human()
```



Creating Python Objects from a class

PowerShell objects are created using the `New-Object` cmdlet, followed by the name of the class.

```
$prateek = New-Object Human

# alternatively
$john = [Human]::new()
```

```
PS C:\Demo>
class Human {
    # this is a property/attribute
    [String]$name = 'homo sapiens'
    [Int] $height = 5
    # define a method/function of a class
    eat(){
        Write-Host $this.name 'is eating now'
    }
}

PS C:\Demo> New-Object Human

name         height
----         ------
homo sapiens      5


PS C:\Demo> [Human]::new()

name         height
----         ------
homo sapiens      5


PS C:\Demo>
```

Creating PowerShell Objects from a class

Accessing Attributes and Functions of an Objects

In order to access the attributes and functions of an object, both PowerShell and Python use a Dot (.) operator. That means, to access the `height` property or `talk()` method of object 'John' in PowerShell, you would write some like this:

```
1 $john.height
2 $john.talk()
```

and almost similar usage in Python, where you can instantiate a class then access and change the attributes using the 'Dot Operator' or even call the methods of the class.

```
>>> john = Human()
>>> dir(john)

['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__\
', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass_\
_', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_e\
x__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__wea\
kref__', 'eat', 'height', 'name']
>>>
>>> john.name
'homo sapiens'
>>> john.name = 'John Wayne'
>>> john.name
'John Wayne'
>>> john.eat()
John Wayne is eating now
>>>
```

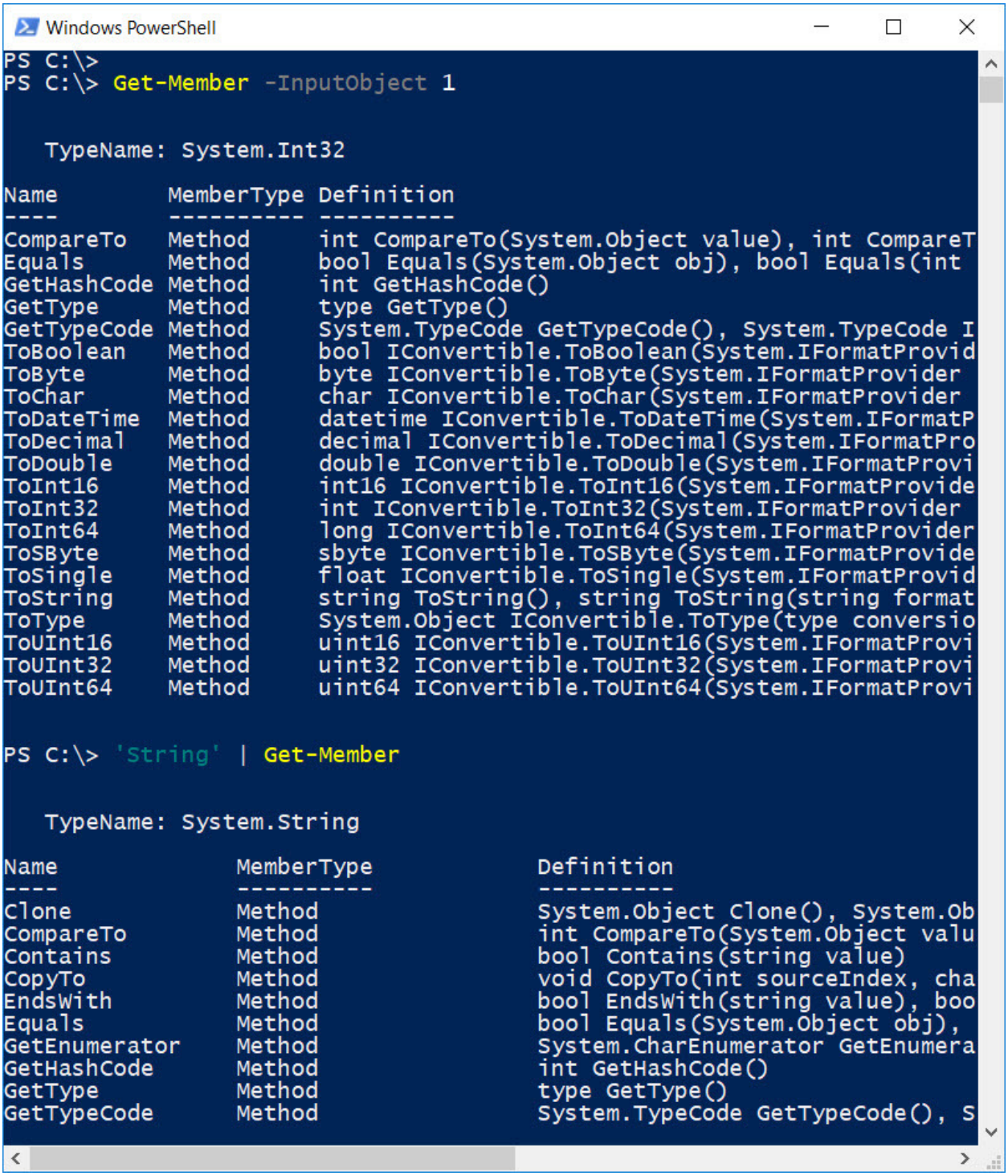
Up to this point in the chapter, we have learned about classes, objects and how to access the members of an object. Now let us look into some ways to inspect objects in PowerShell and Python, also known as Object Introspection. The purpose is to understand what are the members of an object, i.e. Methods and Properties.

'Get-Member' cmdlet in PowerShell

In PowerShell to introspect members of an object we pass the object to the `Get-Member` cmdlet directly or through the pipeline.

Methods and Properties of an object are also called as '`Members`' of the object, this is the reason behind the name of the cmdlet, i.e. `'Get' (Verb) 'Member' (Noun)` as per the PowerShell's 'Verb-Noun' syntax of a cmdlet. `Get-Member` cmdlet will return the member(s) of an object passed to it.

```
# object as an argument to the input object parameter
Get-Member -InputObject 1

# object through the pipeline
"String" | Get-Member
```

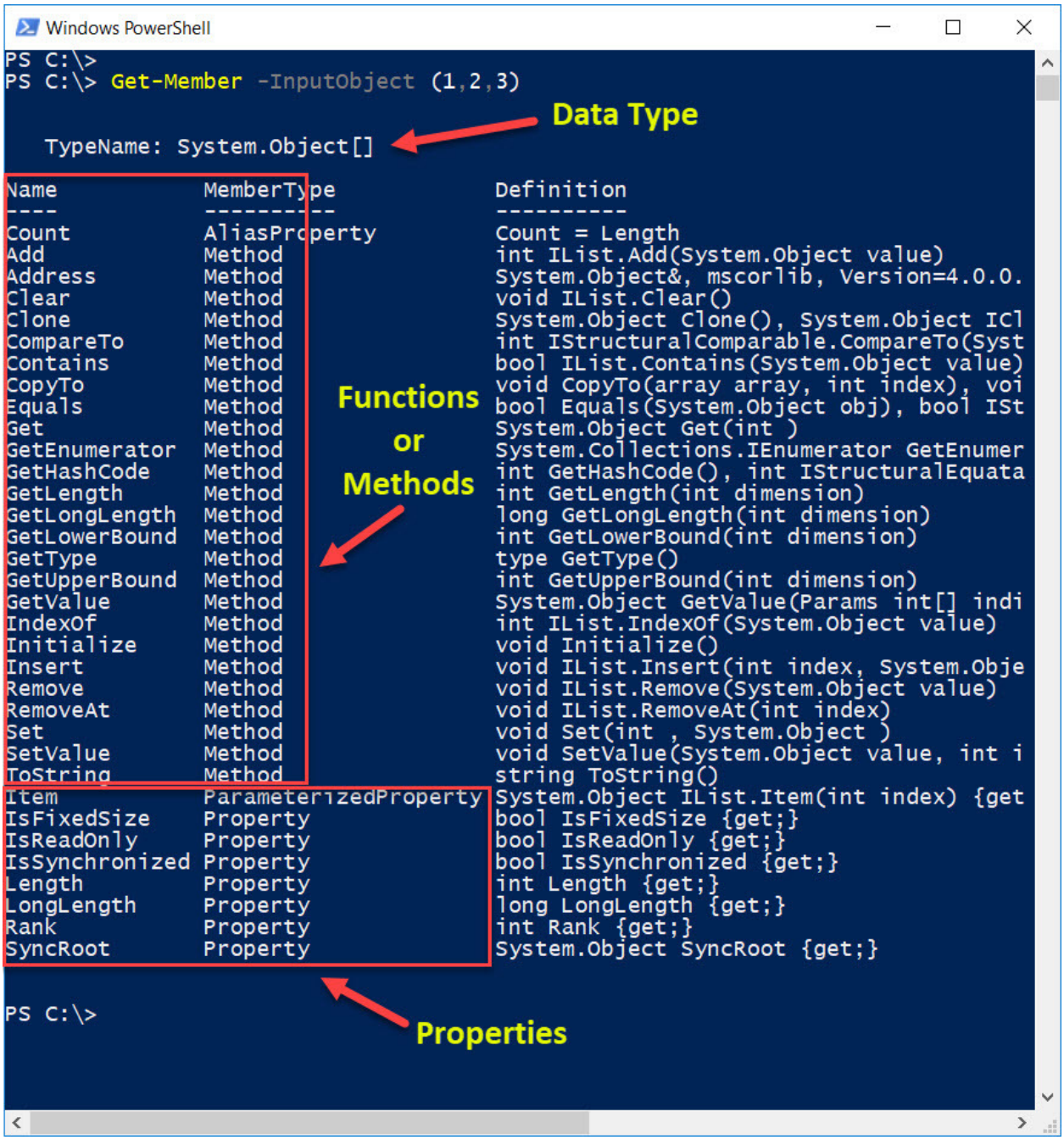


If you look closely in the output of `Get-Member` cmdlet, first you'll notice a '`TypeName`' which is the data type of the object passed to the cmdlet, like `System.String`, `System.Int32` or `System.Object[]`.

Next are properties, with `MemberType` as '`Property`' which is a characteristic of the object like `Rank`, `Count`, `Length` and `IsReadOnly`, and finally the '`Methods`' of the object like `Add()`, `Remove()`

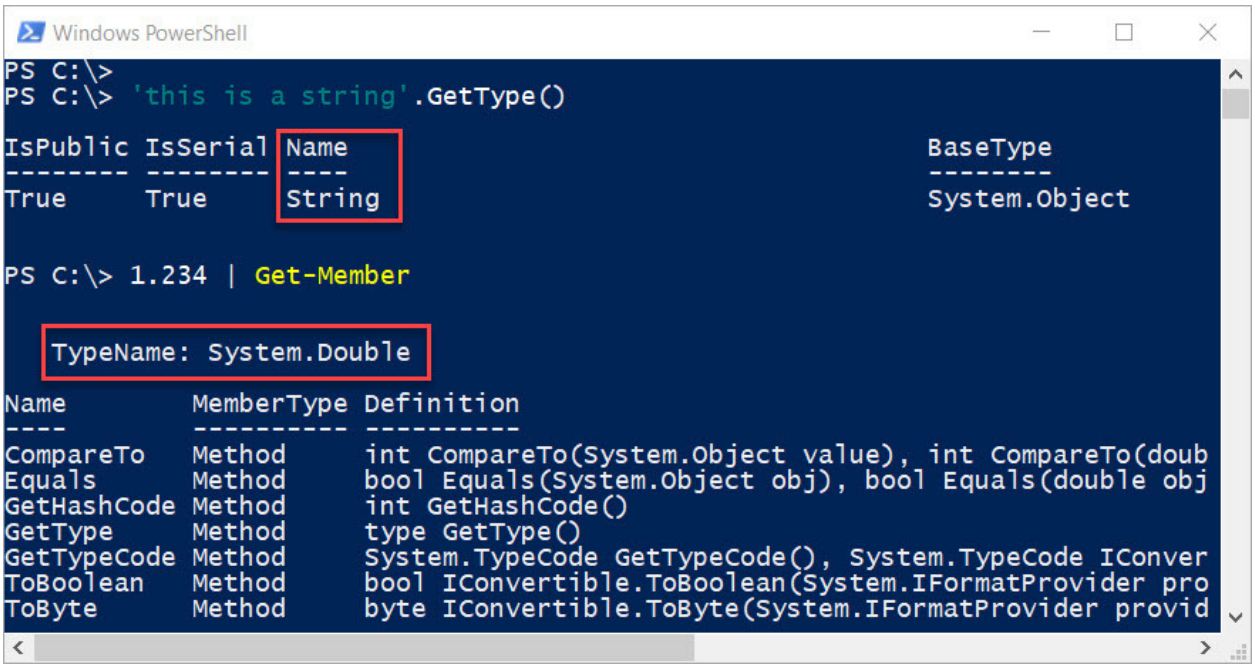
or `ToString()` that represent functionalities that can be performed on the object.

Inspecting the object in PowerShell provides an understanding of properties and methods available in the object.

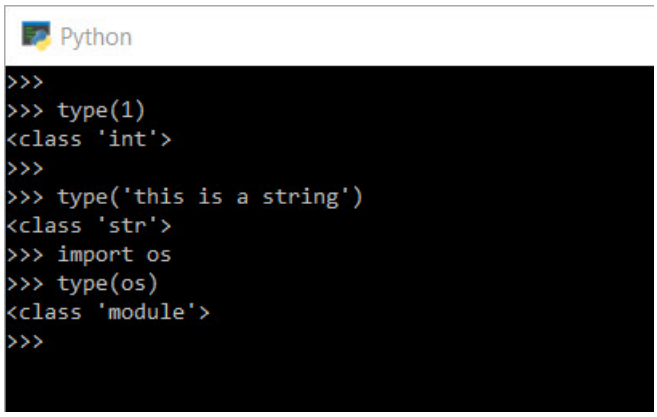


‘type()’ Method in Python

PowerShell returns the `TypeName` or in other words the data type of the object, when an object is passed to the `Get-Member` cmdlet or when you use `GetType()` method of the object.



Python on other hand has a Built-in method called `type()` which returns the data type of the object passed.



‘dir()’ Method in Python

Python has a built-in function `dir()`, that lists all the attributes and methods of an object. Following is the syntax of this function:

```
dir([object])
```

The `dir()` function tries to return a list of valid attributes of the object, but the object we pass as an argument to this function is totally optional. Hence the argument in the above syntax is in brackets (`[ ]`), which implies: it is an option, not mandatory.

`dir()` function when called without any argument will return the names in the current scope.

```
chars = ['a', 'b', 'c']
# with arguments
dir(chars)

# without arguments
dir()
```

```
Python
>>> chars = ['a', 'b', 'c']
>>> # with arguments
... dir(chars)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '_
', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__ge
_', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_sub
_', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__ne
'__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '_
tr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'appen
lear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 're
, 'sort']
>>>
>>> # without arguments
... dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
kage__', '__spec__', 'chars']
>>>
```

dir() function with\without argument

The `dir()` function behaves differently with a different type of object as an argument, to produce the most relevant information. You can basically use this function in 3 different ways by passing Class objects, Module objects and no argument:

1. **Class Objects**: - Returns a list of names of all the valid attributes. Among the use cases of `dir()` function, the main is day to day debugging of programs. The capability of `dir()` is to list out all the attributes of the object passed as an argument, which turn out to be very useful when you are trying to understand the objects you are dealing with in a Python program.

```
 >>> dir(1)
 ['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__\
', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floord\
iv__', '__format__', '__ge__', '__getattribute__', '__getnewargs__', '__gt__', '__ha\
sh__', '__index__', '__init__', '__init_subclass__', '__int__', '__invert__', '__le_\
_', '__lshift__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '_\
_or__', '__pos__', '__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '\
__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', \
'__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__', '__rtru\
ediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclassh\
ook__', '__truediv__', '__trunc__', '__xor__', 'bit_length', 'conjugate', 'denominat\
or', 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']

 >>> dir('a string')
 ['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq\
__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '_\
_gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__'\
, '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__'\
, '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__sub\
classhook__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'ex\
pandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal\
', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'ist\
itle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'rep\
lace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'split\
lines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

2. **Modules Objects** - Returns a list of names of all the attributes, contained in that module. Not only it limits to objects, but you can also list out all the available attributes for a module, which can prove crucial in understanding a module better when you have very little or no information about the module.

```
    >>> import os
    >>> dir(os)
    ['DirEntry', 'F_OK', 'MutableMapping', 'O_APPEND', 'O_BINARY', 'O_CREAT', 'O_EXC\
L', 'O_NOINHERIT', 'O_RANDOM', 'O_RDONLY', 'O_RDWR', 'O_SEQUENTIAL', 'O_SHORT_LIVED'\
, 'O_TEMPORARY', 'O_TEXT', 'O_TRUNC', 'O_WRONLY', 'P_DETACH', 'P_NOWAIT', 'P_NOWAITO\
', 'P_OVERLAY', 'P_WAIT', 'PathLike', 'R_OK', 'SEEK_CUR', 'SEEK_END', 'SEEK_SET', 'T\
MP_MAX', 'W_OK', 'X_OK', '_Environ', '__all__', '__builtins__', '__cached__', '__doc\
__', '__file__', '__loader__', '__name__', '__package__', '__spec__', '_execvpe', '_\
exists', '_exit', '_fspath', '_get_exports_list', '_putenv', '_unsetenv', '_wrap_clo\
se', 'abc', 'abort', 'access', 'altsep', 'chdir', 'chmod', 'close', 'closerange', 'c\
pu_count', 'curdir', 'defpath', 'device_encoding', 'devnull', 'dup', 'dup2', 'enviro\
n', 'errno', 'error', 'execl', 'execle', 'execlp', 'execlpe', 'execv', 'execve', 'ex\
```

```
ecvp', 'execvpe', 'extsep', 'fdopen', 'fsdecode', 'fsencode', 'fspath', 'fstat', 'fs\
ync', 'ftruncate', 'get_exec_path', 'get_handle_inheritable', 'get_inheritable', 'ge\
t_terminal_size', 'getcwd', 'getcwdb', 'getenv', 'getlogin', 'getpid', 'getppid', 'i\
satty', 'kill', 'linesep', 'link', 'listdir', 'lseek', 'lstat', 'makedirs', 'mkdir',\
 'name', 'open', 'pardir', 'path', 'pathsep', 'pipe', 'popen', 'putenv', 'read', 're\
adlink', 'remove', 'removedirs', 'rename', 'renames', 'replace', 'rmdir', 'scandir',\
 'sep', 'set_handle_inheritable', 'set_inheritable', 'spawnl', 'spawnle', 'spawnv', \
'spawnve', 'st', 'startfile', 'stat', 'stat_float_times', 'stat_result', 'statvfs_re\
sult', 'strerror', 'supports_bytes_environ', 'supports_dir_fd', 'supports_effective_\
ids', 'supports_fd', 'supports_follow_symlinks', 'symlink', 'sys', 'system', 'termin\
al_size', 'times', 'times_result', 'truncate', 'umask', 'uname_result', 'unlink', 'u\
random', 'utime', 'waitpid', 'walk', 'write']
```

3. **No Object** - In case the object is not passed to the function as an argument, it returns a list of names in the current local scope.

```
    >>> dir()
    ['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__pack\
age__', '__spec__', 'chars', 'os']
```

'inspect' Module in Python

Python is also equipped with a library called 'inspect', that can be utilized to get the members of live Python objects, even during the program execution.

Following are some of the useful methods provided by this module that can check if any object or any members of the object is a module, class, function etc. on basis of the Boolean output '`True`' or '`False`' :

`ismodule()`

`isclass()`

`ismethod()`

`isfunction()`

`isgeneratorfunction()`

`isgenerator()`

`istraceback()`

`isframe()`

`iscode()`

`isbuiltin()`

For example, if we pass the `os` module to the `ismodule()` method you'll get `True` as the output, whereas when we pass another object let's suppose a string you'll get the output as `False`.

```
>>> import os, inspect
>>> inspect.ismodule(os)
True
>>> inspect.ismodule('this is a string')
False
```

One of my favorite use of `inspect` module is with the `getmembers()` method that returns all the members of the object that satisfy a given condition. For example, if I want to find only the built-in methods of the object I can use the named parameter: '`predicate`' of the `getmembers()` method to filter that out, as demonstrated in the following code sample:

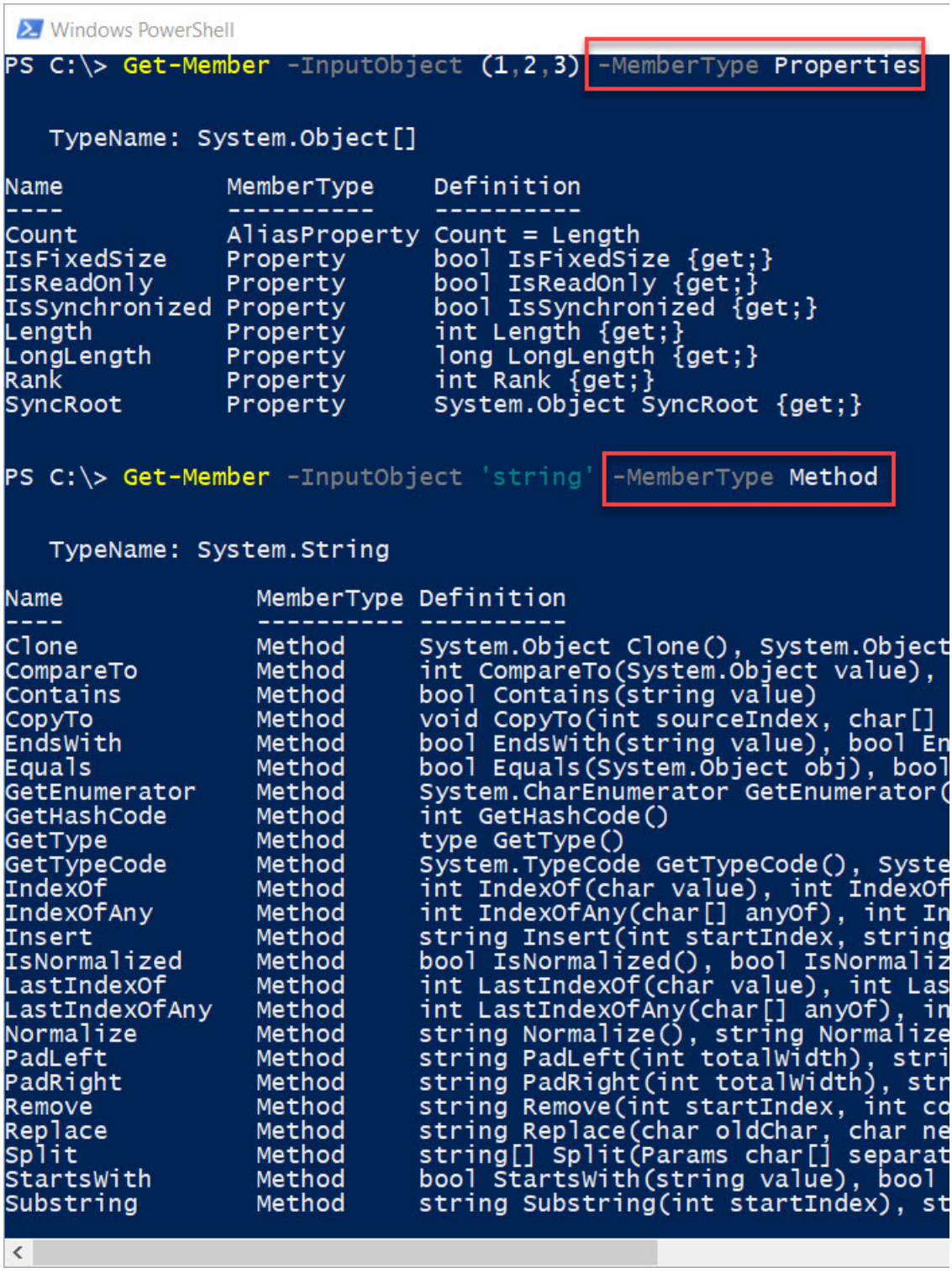
```
# import inspect module
import inspect

# define a float variable
pi = 3.14159

# find the members of the object categorized as built-in methods
inspect.getmembers(pi, predicate=inspect.isbuiltin)

# once you know the mattributes/methods of the object, you can use them
pi.is_integer()
pi.as_integer_ratio()
```

To make the output a little bit easy to understand and readable, I'll enumerate the members and print them separately using a `for` loop. No need to worry if you are not yet familiar with concepts and working of `for` loops in Python, as there are multiple opportunities in later chapters of this book, that will explain `for` loops in detail with examples.

```
>>> for member in inspect.getmembers(pi, predicate=inspect.isbuiltin) :
...     print(member)
```

getmember() method in Python

On another hand in PowerShell, methods and properties of an object are filtered out using the `Get-Member` cmdlet, by defining the `-MemberType` parameter.



Filtering Member Types in PowerShell

To know more about the '`inspect`' module and its members, you can also use the `dir()` built-in function, explained in one of the previous sub-sections of this chapter, to return the list of members.

Where you'll also find some of the methods we have described in this sub-section, like `ismodule()`, `is_integer()` etc.

```
>>> import inspect
>>> dir(inspect)
['ArgInfo', 'ArgSpec', 'Arguments', 'Attribute', 'BlockFinder', 'BoundArguments', 'C\
ORO_CLOSED', 'CORO_CREATED', 'CORO_RUNNING', 'CORO_SUSPENDED', 'CO_ASYNC_GENERATOR',\
 'CO_COROUTINE', 'CO_GENERATOR', 'CO_ITERABLE_COROUTINE', 'CO_NESTED', 'CO_NEWLOCALS\
', 'CO_NOFREE', 'CO_OPTIMIZED', 'CO_VARARGS', 'CO_VARKEYWORDS', 'ClosureVars', 'EndO\
fBlock', 'FrameInfo', 'FullArgSpec', 'GEN_CLOSED', 'GEN_CREATED', 'GEN_RUNNING', 'GE\
N_SUSPENDED', 'OrderedDict', 'Parameter', 'Signature', 'TPFLAGS_IS_ABSTRACT', 'Trace\
back', '_ClassMethodWrapper', '_KEYWORD_ONLY', '_MethodWrapper', '_NonUserDefinedCal\
lables', '_POSITIONAL_ONLY', '_POSITIONAL_OR_KEYWORD', '_ParameterKind', '_VAR_KEYWO\
RD', '_VAR_POSITIONAL', '_WrapperDescriptor', '__author__', '__builtins__', '__cache\
d__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', '_\
check_class', '_check_instance', '_empty', '_filesbymodname', '_findclass', '_finddo\
c', '_getfullargs', '_is_type', '_main', '_missing_arguments', '_sentinel', '_shadow\
ed_dict', '_signature_bound_method', '_signature_from_builtin', '_signature_from_cal\
lable', '_signature_from_function', '_signature_fromstr', '_signature_get_bound_para\
m', '_signature_get_partial', '_signature_get_user_defined_method', '_signature_is_b\
uiltin', '_signature_is_functionlike', '_signature_strip_non_python_syntax', '_stati\
c_getmro', '_too_many', '_void', 'ast', 'attrgetter', 'builtins', 'classify_class_at\
trs', 'cleandoc', 'collections', 'currentframe', 'dis', 'enum', 'findsource', 'forma\
tannotation', 'formatannotationrelativeto', 'formatargspec', 'formatargvalues', 'fun\
ctools', 'getabsfile', 'getargs', 'getargspec', 'getargvalues', 'getattr_static', 'g\
etblock', 'getcallargs', 'getclasstree', 'getclosurevars', 'getcomments', 'getcorout\
inelocals', 'getcoroutinestate', 'getdoc', 'getfile', 'getframeinfo', 'getfullargspe\
c', 'getgeneratorlocals', 'getgeneratorstate', 'getinnerframes', 'getlineno', 'getme\
mbers', 'getmodule', 'getmodulename', 'getmro', 'getouterframes', 'getsource', 'gets\
ourcefile', 'getsourcelines', 'importlib', 'indentsize', 'isabstract', 'isasyncgen',\
 'isasyncgenfunction', 'isawaitable', 'isbuiltin', 'isclass', 'iscode', 'iscoroutine\
', 'iscoroutinefunction', 'isdatadescriptor', 'isframe', 'isfunction', 'isgenerator'\
, 'isgeneratorfunction', 'isgetsetdescriptor', 'ismemberdescriptor', 'ismethod', 'is\
methoddescriptor', 'ismodule', 'isroutine', 'istraceback', 'itertools', 'k', 'lineca\
che', 'mod_dict', 'modulesbyfile', 'namedtuple', 'os', 're', 'signature', 'stack', '\
sys', 'token', 'tokenize', 'trace', 'types', 'unwrap', 'v', 'walktree', 'warnings']
```

Key Pointers

- Class is a category, set or a classification that has some attribute in common and can be differentiated from other classes by kind, type, or quality

Python Syntax:

```
class human:
    # body of the class
```

PowerShell Syntax:

```
class human{
    # body of the class
}
```

- Objects are instances of a class and have all methods and properties defined in the class.

Python Objects:

```
# returns a Python object of 'Human' Class
Human()
```

PowerShell Objects:

```
# returns a PowerShell object of 'Human' Class
New-Object Human

# alternatively
[Human]::new()
```

- All Attributes and Functions of an object in both PowerShell and Python is accessible using the Dot (`.`) operator.
- `Get-Member` cmdlet in PowerShell will return the properties and methods of an object passed to it. Python on other hand offers a `dir()` function, that return members of an object passed to it.
- Python has a Built-in method called `type()` which returns the data type of the object passed, this is exactly how you get the data type of an object in PowerShell using the `GetType()` method.
- 'inspect' module in Python, also has the capabilities to return the members of live Python objects, even during the program execution. It enables developers to filter the members of the object by their type and check the type of object.

Reading Recommendations

To learn more, I'd advise you to please go through following PowerShell and Python help documentation from the console and/or find links to some additional external reading resources.

PowerShell Help

- `Get-Help about_Classes`

- `Get-Help about_Objects`
- `Get-Help about_Properties`
- `Get-Help about_Methods`
- `Get-Help Get-Member`

Python Help

- `help('class')`
- `help(dir)`
- `dir(inspect)`
- `import inspect; help(inspect)`

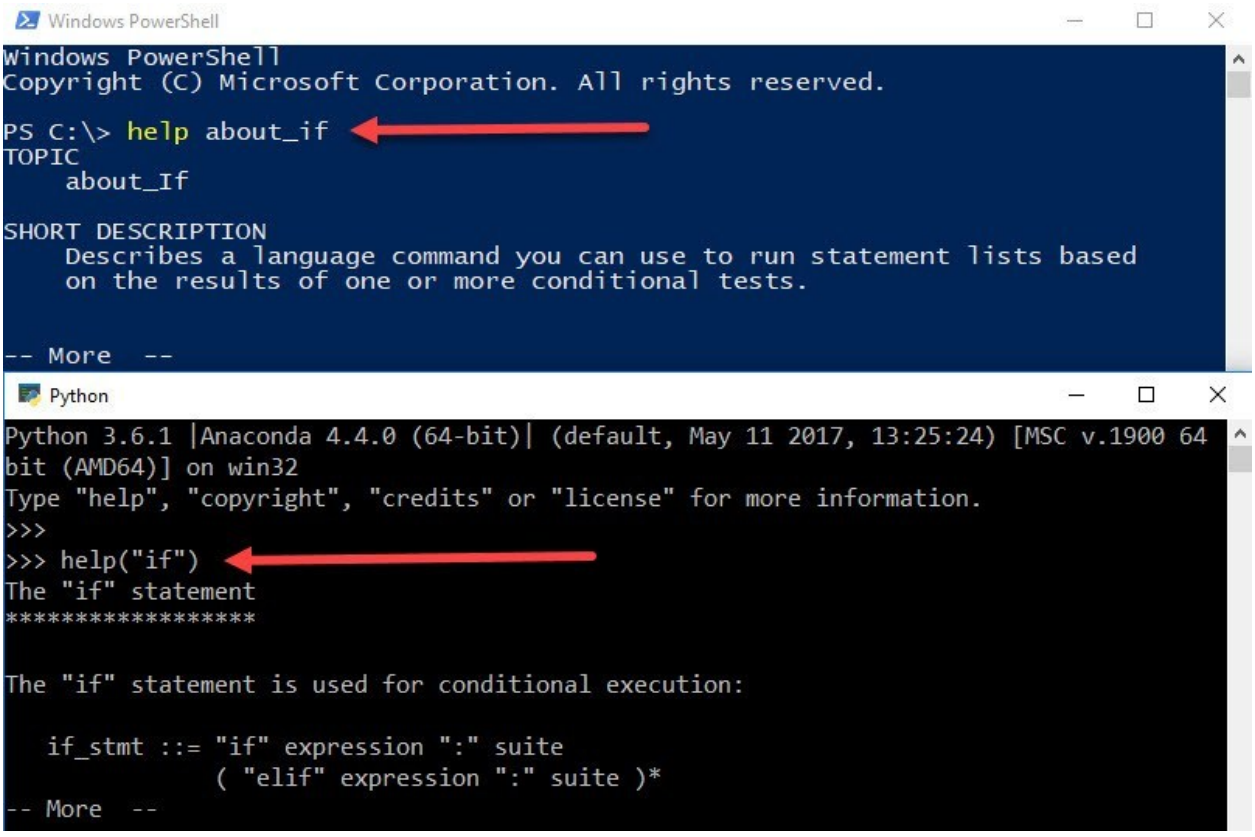
Links

- Method Vs Functions[a]
- Inspect Live Objects[b]

[a] https://stackoverflow.com/questions/155609/whats-the-difference-between-a-method-and-a-function
[b] https://docs.python.org/3/library/inspect.html

Chapter 6 - The Help System

Python and PowerShell are both equipped with inbuilt help system, so that developers and users can easily understand the concept, modules and and elements of scripting language. In built help system also helps in introspection of various object types and understanding how a module and program will work.

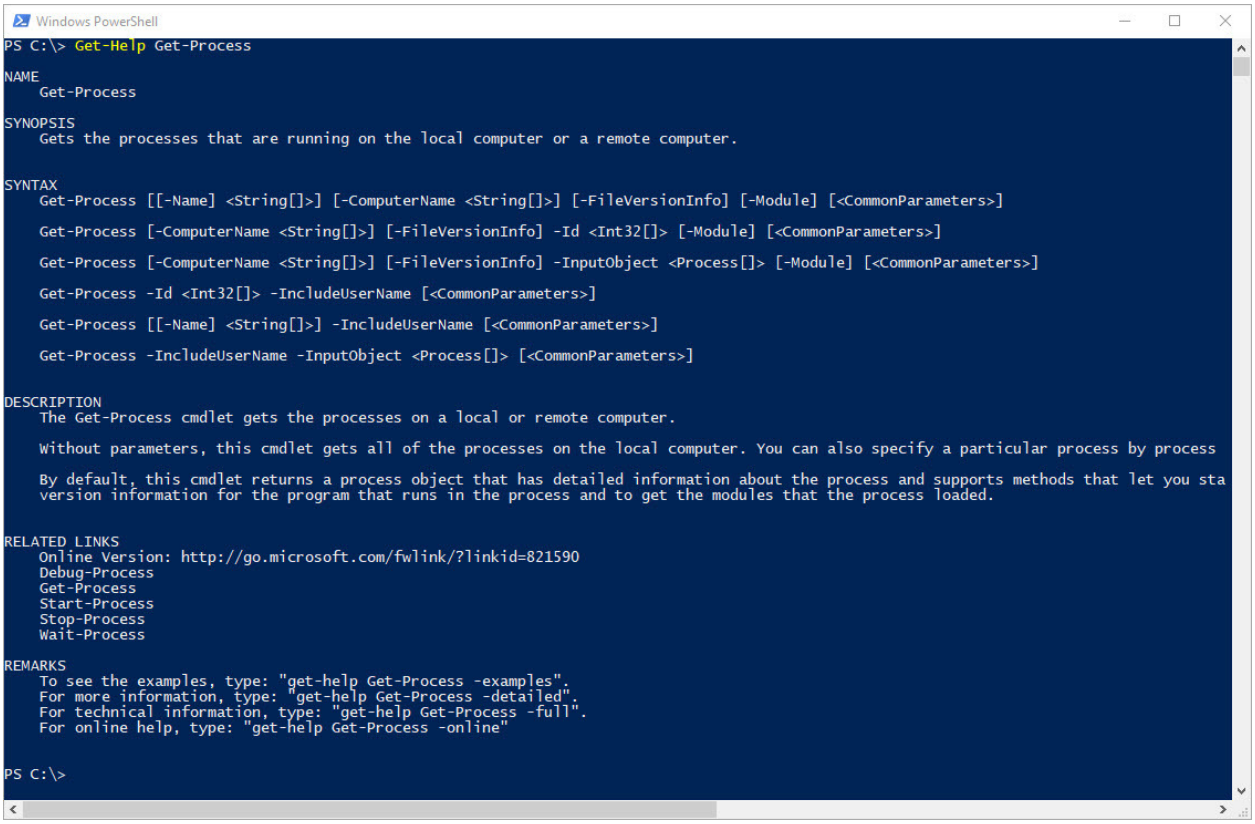


Get-Help cmdlet in PowerShell

PowerShell provides a very robust help system through the `Get-Help` cmdlet, where you can read the help documents on an array of topics and concepts. The `Get-Help` cmdlet can be run against strings and keywords with wildcards, to return detailed help documentation, which can be referred for examples and use cases to understand how PowerShell work, or explains concepts, including the elements of the Windows PowerShell language.

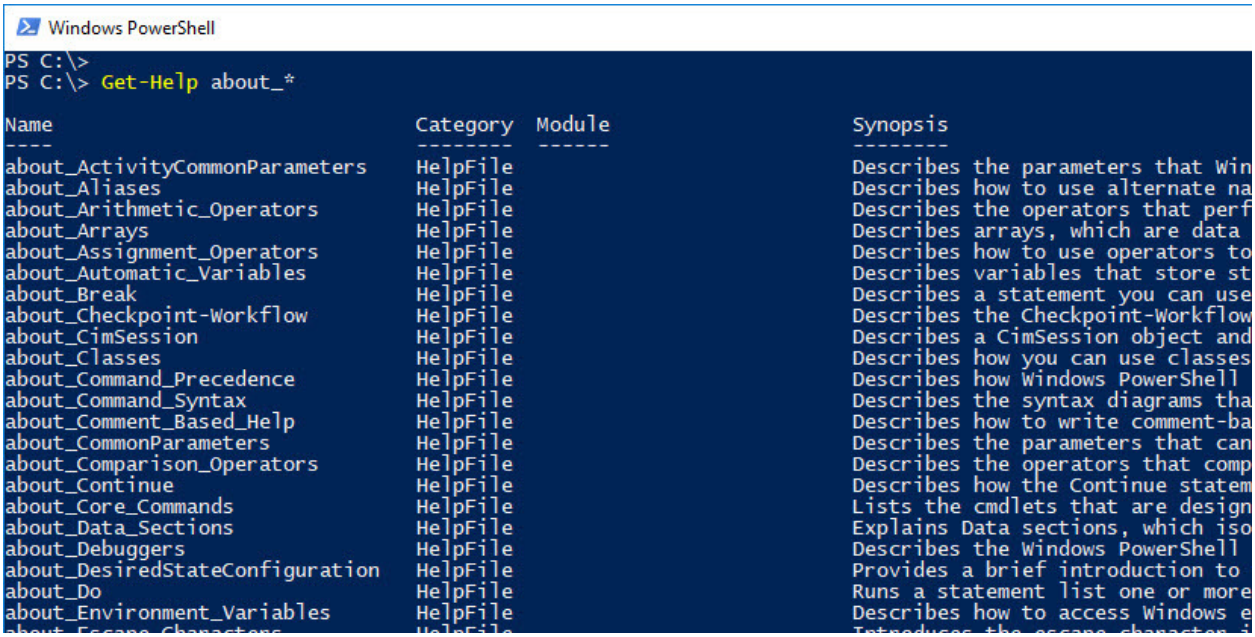
Following code sample return the help document for the `Get-Process` cmdlet

```
Get-Help Get-Process

# use the -Full switch parameter to return complete documentation
Get-Help Get-Process -Full
```



For more conceptual topics run the help with wildcards like `about_*`

```
Get-Help About_*
```



‘help()’ Method in Python

Python has a built-in help system that is accessible through built-in method `help()` and this method can be called interactively with maximum one object as an argument, in the following syntax to return the help documents in Python.

```
help(object)
```

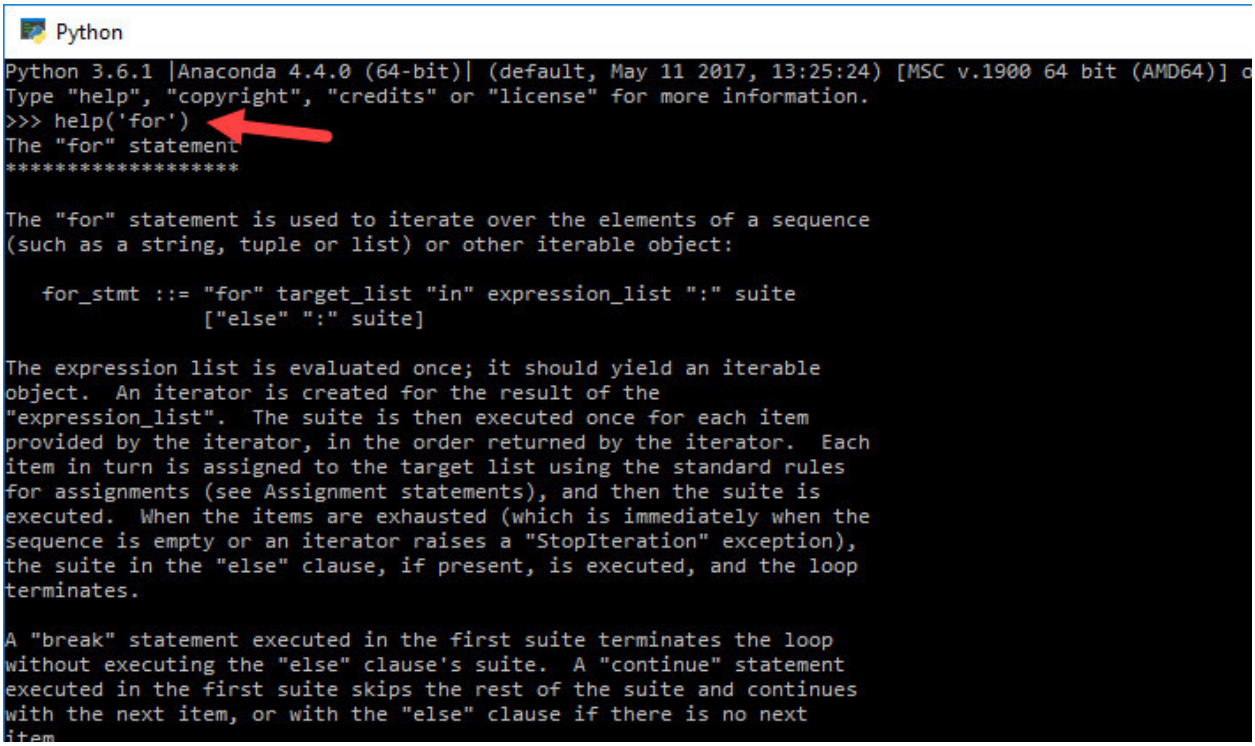
You can try following method calls and see the results yourself.

```
help([1,2,3]) # python [list] as argument
help(3.14) # float number as argument
help(help) # help on help
help("builtin") # list builtin methods
help("modules") # list modules
help(dict) # help on dictionaries
help(int) # help on integer
help(bool) # help on boolean
help('def') # any topic as argument
```

String Argument passed to help() method

When a string is passed to the `help()` method then it is looked up as keyword, function, class, name of a module or a documentation topic, like in the following example help documentation of ‘`for`’ statement is invoked.

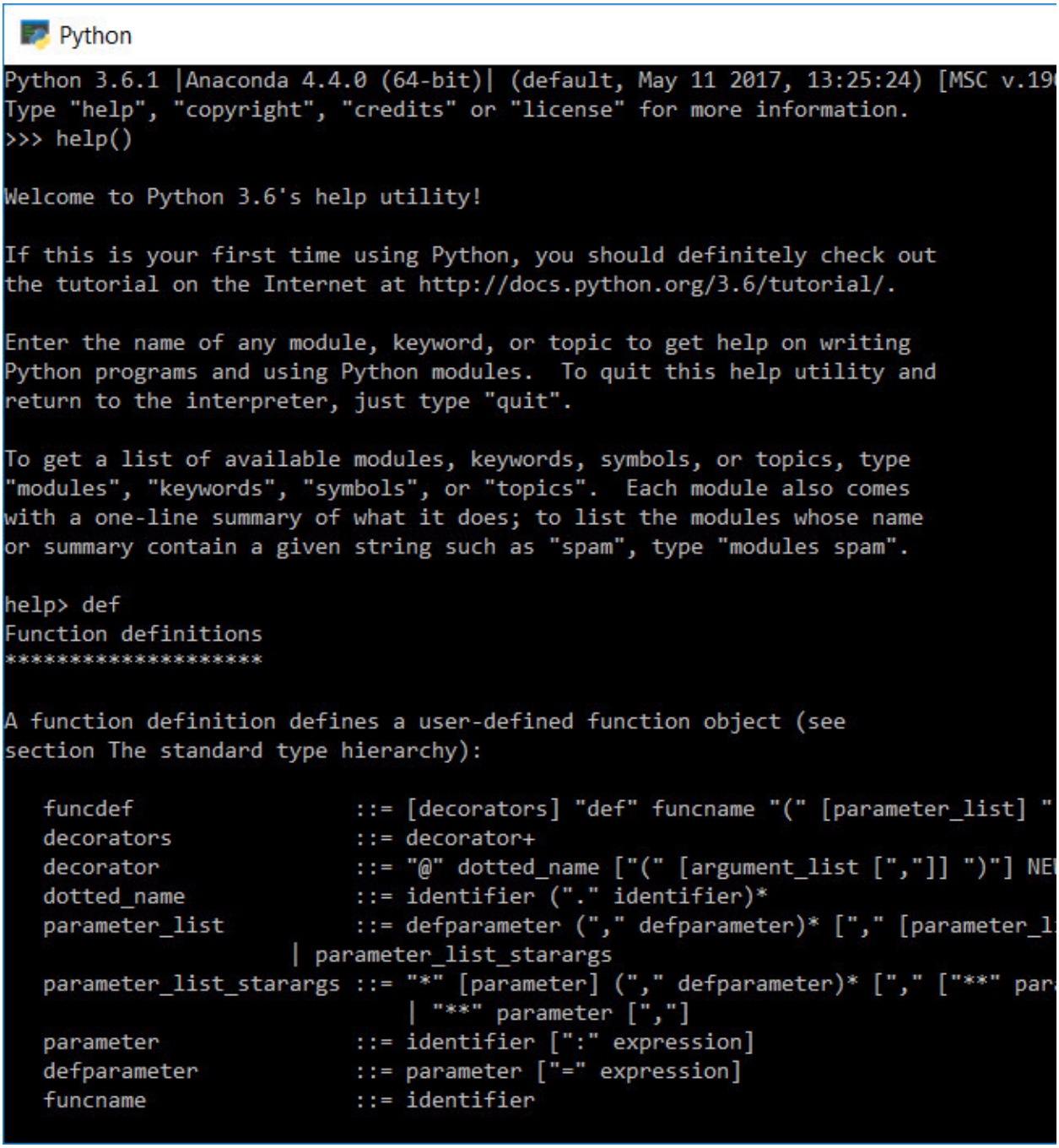
```
1 help('for')
```



No Argument passed to help() method

In case no argument is passed to the built-in method `help()` then interactive help system is launched in your Python console. Where you can directly provide the name of topic to get help on followed by an enter key.

```
1 help() # without any arguments
```



Understanding How a Module Works

In Python is object oriented programming language, even the modules are treated as objects. So when you import any module in the current session you can also pass it the `help(<object>)` method to see the help documentation.

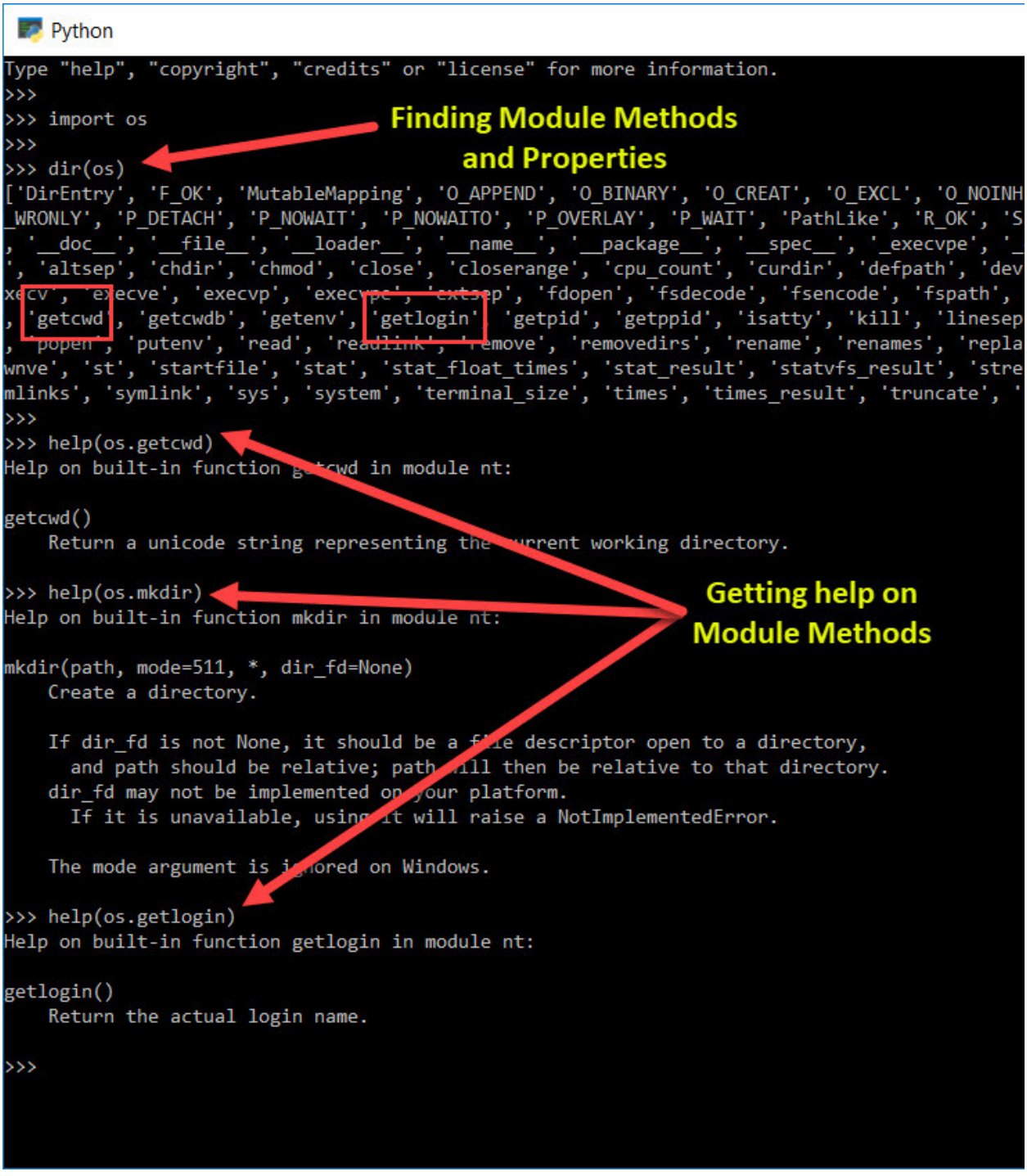
```
1 import sys
2 help(sys)
```

Python

```
Python 3.6.1 |Anaconda 4.4.0 (64-bit)| (default, May 11 2017, 13:25:24) [MSC v.1900 64
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> import sys
>>>
>>> help(sys)
Help on built-in module sys:

NAME
    sys

MODULE REFERENCE
    https://docs.python.org/3.6/library/sys

    The following documentation is automatically generated from the Python
    source files.  It may be incomplete, incorrect or include features that
    are considered implementation detail and may vary between Python
    implementations.  When in doubt, consult the module reference at the
    location listed above.

DESCRIPTION
    This module provides access to some objects used or maintained by the
    interpreter and to functions that interact strongly with the interpreter.

    Dynamic objects:

    argv -- command line arguments; argv[0] is the script pathname if known
    path -- module search path; path[0] is the script directory, else ''
    modules -- dictionary of loaded modules

    displayhook -- called to show results in an interactive session
    excepthook -- called to handle any uncaught exception other than SystemExit
      To customize printing in an interactive session or to install a custom
      top-level exception handler, assign other functions to replace these.

    stdin -- standard input file object; used by input()
    stdout -- standard output file object; used by print()
```

Since an object has Methods and Properties as members, you can also get help on a modules members: methods and properties. In previous chapter 'Object Introspection' we looked into this, where I explained how to find an object's members using the `dir()` method, once you have them just pass the object with a (".") Dot operator, followed by the property name to the `help(<object.property>)` method to a get help description.

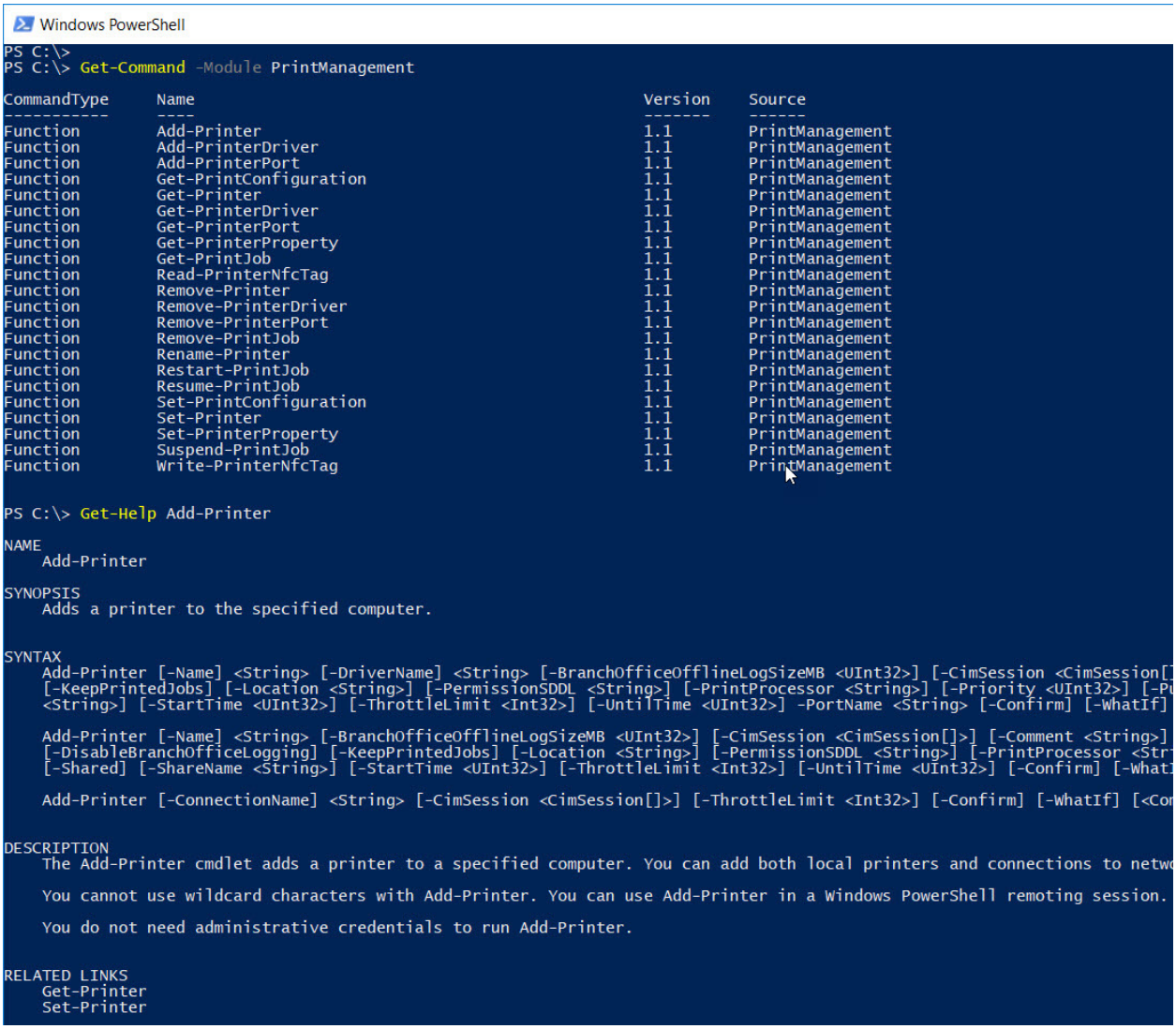
```
import os
# list the members of the object
dir(os)
# help of module methods
help(os.getcwd)
help(os.mkdir)
help(os.getlogin)
```




```
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> import os
>>>
>>> dir(os)
['DirEntry', 'F_OK', 'MutableMapping', 'O_APPEND', 'O_BINARY', 'O_CREAT', 'O_EXCL', 'O_NOINH
_WRONLY', 'P_DETACH', 'P_NOWAIT', 'P_NOWAITO', 'P_OVERLAY', 'P_WAIT', 'PathLike', 'R_OK', 'S
, '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', '_execvpe', '_
', 'altsep', 'chdir', 'chmod', 'close', 'closerange', 'cpu_count', 'curdir', 'defpath', 'dev
xecv', 'execve', 'execvp', 'execvpe', 'extsep', 'fdopen', 'fsdecode', 'fsencode', 'fspath',
, 'getcwd', 'getcwdb', 'getenv', 'getlogin', 'getpid', 'getppid', 'isatty', 'kill', 'linesep
, 'popen', 'putenv', 'read', 'readlink', 'remove', 'removedirs', 'rename', 'renames', 'repla
wnve', 'st', 'startfile', 'stat', 'stat_float_times', 'stat_result', 'statvfs_result', 'stre
mlinks', 'symlink', 'sys', 'system', 'terminal_size', 'times', 'times_result', 'truncate', '
>>>
>>> help(os.getcwd)
Help on built-in function getcwd in module nt:

getcwd()
    Return a unicode string representing the current working directory.

>>> help(os.mkdir)
Help on built-in function mkdir in module nt:

mkdir(path, mode=511, *, dir_fd=None)
    Create a directory.

    If dir_fd is not None, it should be a file descriptor open to a directory,
      and path should be relative; path will then be relative to that directory.
    dir_fd may not be implemented on your platform.
      If it is unavailable, using it will raise a NotImplementedError.

    The mode argument is ignored on Windows.

>>> help(os.getlogin)
Help on built-in function getlogin in module nt:

getlogin()
    Return the actual login name.

>>>
```

This is exactly how you find cmdlets of a PowerShell Module using the `Get-Command -Module <Module Name>` and then their help description using `Get-Help <Cmdlet Name>` cmdlet

```
# list all cmdlets from the module
Get-Command -Module PrintManagement
# get the help documentation of the module cmdlet
Get-Help Add-Printer
```



Chapter 7 - Modules

In simple terms, a module is a file consisting of Python or PowerShell code which has functions, classes and variables definitions inside it.

Modular Programming

A module can also be referred a a logical organization of your code, and the process of breaking large and complex programming tasks into separate, small and manageable subtasks or modules is called Modular Programming. These small subtasks can be coupled together to form building blocks of a complex but neat programming system.

Modular programming brings has several advantages like, Simplicity, Reusability and Maintainability in large programs.

Importing Modules

Importing Single Module

PowerShell provides an `Import-module` cmdlet to import modules in the current session.

```
# syntax: Import-Module <name of module>
Import-Module ActiveDirectory
```

Whereas, Python has a keyword `import` to load modules into the current Python Session, but the `import` statement does not make contents in the module directly accessible to the caller and are only accessible when they are prefixed with `<module name>` followed by a (‘ . ‘) dot notation: `<module name>.<property/method>` like in the following example.

```
import os # importing module
print(os.getcwd()) # gets current working directory
```

Importing Multiple Modules

Both PowerShell and Python allows you to import multiple modules in the session by referencing the module names separated by (‘ , ‘) comma(s), for example in Python the syntax would look something like `import <module1>, <module2> ...`

```
import time, sys # importing multiple modules
time.sleep(2) # sleeps for 2 seconds
print("Platform:",sys.platform) # platform identifier, like Win32
```

Similarly, in PowerShell `Import-Module` cmdlet also follows almost the same syntax: `Import-Module <module1>, <module2> ...`

```
Import-Module PSReadLine, AzureRM, Posh-Git
```

Importing Sub-Modules

Python allows us to import only the sub-module(s) of a module, which are basically packages inside a Python module using the `from ... import ...` keywords.

```
# SYNTAX: from <module>import <submodule1, sunmodule2, ...>
from random import shuffle
array = [1,2,3,4]
shuffle(array)
print('shuffled array:', array)
```

or importing multiple sub-modules separated by commas.

```
from statistics import mean, variance, median, stdev
```

Programmers are also allowed to import all sub modules from a module, using the following syntax: `from <module name> import *`

```
from random import *
```

Module Aliases

Python also allows programmers to define alternate names or aliases of the modules. While importing a module use the `as` keyword in following syntax `import <module name> as <alternate name>` to define an alias or an alternative names, these are easy to use and you don't have to type the complete name of the module while accessing module objects.

```
import math as m
print(m.pi)
print(m.e)
```

Just to summarize everything, following are number of ways to import a module in python:

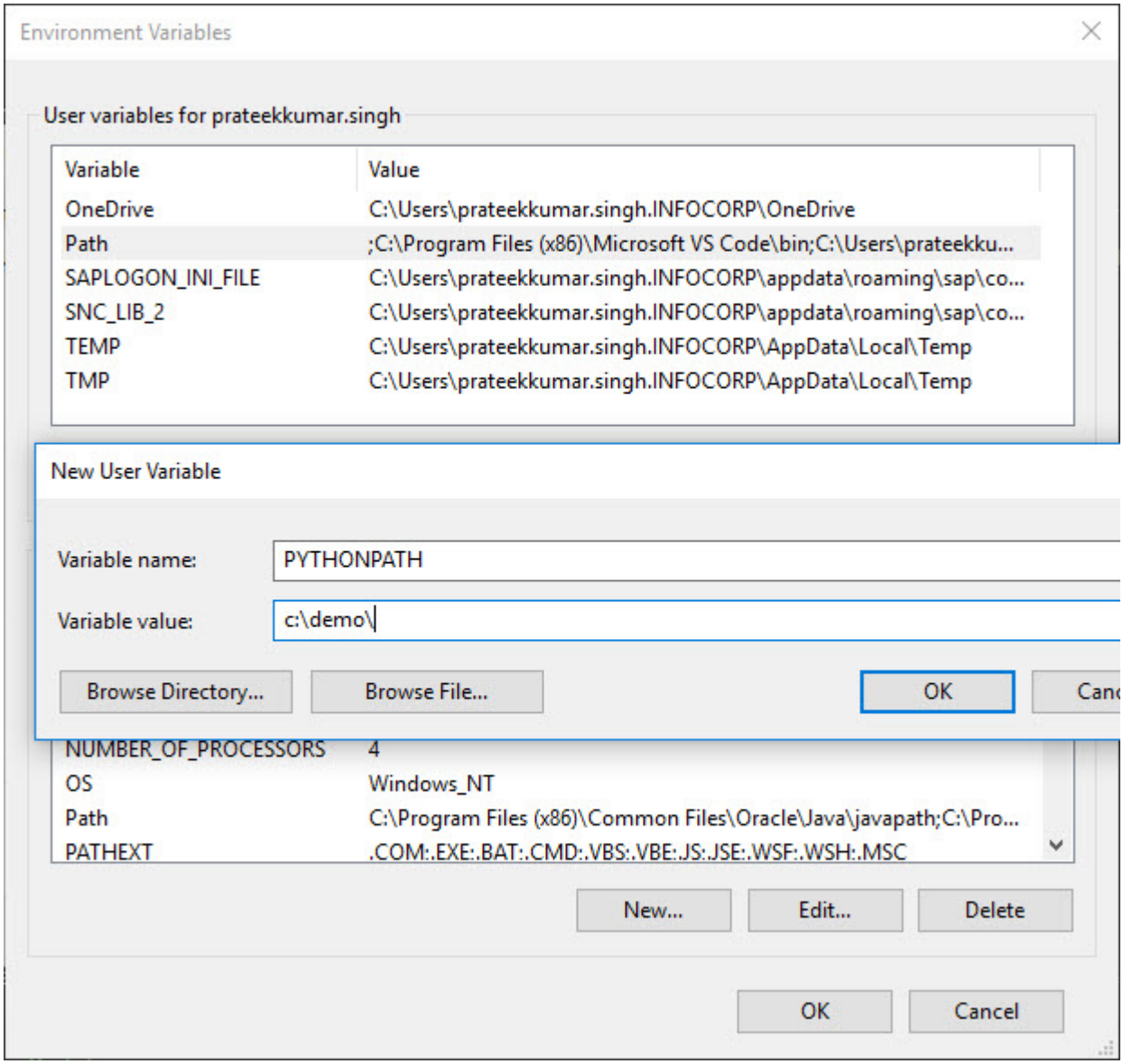
```
import matplotlib
import matplotlib.pyplot # import sub-module: pyplot
import matplotlib.pyplot as plt # import sub-module: pyplot with alias `plt`
from matplotlib import pyplot # import sub-module: pyplot
from matplotlib import pyplot as plt # import sub-module: pyplot with alias `plt`
```

PYTHONPATH and $Env:PSModulePath

In PowerShell `$env:PSModulePath` Environment Variable defines paths to the locations of modules that are installed on the computer, which helps PowerShell to locate the modules on the system while importing them into the PowerShell session.

```
PS C:\data> $env:PSModulePath -split ';'

C:\Users\prateek\Documents\WindowsPowerShell\Modules
C:\Program Files\WindowsPowerShell\Modules
C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules
```

Python allows programmers to create an Environment variable `PYTHONPATH` which is the default search path for module files, and if this variable is set it will mimic the `PATH` environment variable. That means the directories mentioned under the `PYTHONPATH` variable will appear in `PATH` variable, so that Python knows where to lookup module files when requested, like in the following example.




```
import sys
sys.path

['C:\\Users\\prateek\\AppData\\Local\\Continuum\\Anaconda3\\Scripts',
 'C:\\demo',
 'C:\\Users\\prateek\\AppData\\Local\\Continuum\\Anaconda3\\python36.zip',
 'C:\\Users\\prateek\\AppData\\Local\\Continuum\\Anaconda3\\DLLs',
 'C:\\Users\\prateek\\AppData\\Local\\Continuum\\Anaconda3\\lib',
 'C:\\Users\\prateek\\AppData\\Local\\Continuum\\Anaconda3']
```

Often you'll come across scenarios, where you'll want to add the directories/folder paths to variable:

`PATH`, `PYTHONPATH` or `$env:PSModulePath` during the program run time, which can be done in the following ways-

In PowerShell you add a custom module directory path: `c:\ModulePath` by adding it and reassigning the value to the `$env:PSModulePath` environment variable.

```
$env:PSModulePath = $env:PSModulePath + ";c:\ModulePath"
```

But, Python allows you to directly append the folder path to the `PATH` environment variable using the `append()` method of `sys` module, like in the following code sample.

```
import sys
sys.path.append('C:\\data\\modules')
sys.path

['',
 'C:\\data\\modules'
 'C:\\demo',
 'C:\\Users\\prateek\\AppData\\Local\\Continuum\\Anaconda3\\Scripts',
 'C:\\Users\\prateek\\AppData\\Local\\Continuum\\Anaconda3\\python36.zip',
 'C:\\Users\\prateek\\AppData\\Local\\Continuum\\Anaconda3\\DLLs',
 'C:\\Users\\prateek\\AppData\\Local\\Continuum\\Anaconda3\\lib',
 'C:\\Users\\prateek\\AppData\\Local\\Continuum\\Anaconda3',
 ]
```

Installing Modules

One of the best things about using Python and PowerShell is availability of number of fantastic Modules and libraries, that will help avoid doing a lot of coding by yourself and will definiteky save decent amount of time. But before you can even use these Modules and Libraries, you've to first install them on your computer, then only you can import and use them in your session as discussed in previous subsection of this chapter.

There are many ways to install modules on computer, but here we are only discussing about the two major and easy approaches.

Install-Module

PowerShell offers an `Install-Module` cmdlet that can get one or more modules that meet specified criteria from an online gallery and copies module folders to the installation location, for example if you want to install a module named `posh-git`, you can simply run the below command from PowerShell console with administrative privileges.

```
Install-Module -Name posh-git
Install-Module -Name sqlserver
Install-Module -Name AzureRM
```

You can even install multiple modules at once as described in the following example using the commas separating the module names.

```
Install-Module -Name posh-git, sqlserver, AzureRM
```

PIP

`pip` is a 'package management system' program written in Python, which is used to install and manage Python software packages from trusted 3rd-party online software repositories. Following is a command line to install Python libraries using `pip` on a windows machine using the `-m flag` which helps Python to find the `pip` module.

```
# instaling modules in python using PIP Packakge manager
# which downloads and installs packages from an online repository
python.exe -m pip install numpy
python.exe -m pip install tweepy
```

and installing multiple libraries/modules at once.

```
python.exe -m pip install numpy, tweepy
```

Creating Modules

Writing or creating a module is just like creating any other program file in Python or PowerShell. To create a module in Python let's just take below mentioned sample function to print some text and save it in a file named: `hello.py`

```
# function definition
def world():
    print("Hello World! from Python")
```

Now you can directly import this module to your Python session, and call functions, properties defined in the module file.

```
# importing the module
import hello

# function call
hello.world()
```

You'll get the following result on your screen upon execution of above code.

```
"Hello World! from Python"
```

Similarly, PowerShell modules are no different and are just PowerShell scripts saved as an extension: `.psm1` **not** `.ps1`, let take the following function and save it in a file named: `hello.psm1`

```
# function definition
function helloworld(){
    "Hello World! from PowerShell"
}
```

Now you can import this module into a PowerShell session using `Import-Module` cmdlet and run the function defined in the module, like in following example:

```
Import-Module hello
helloworld
```

which will return the string value as expected,

```
Hello World! from PowerShell
```

Chapter 8 - Data Types

Python and PowerShell like any other programming languages, have data types that are used to classify one particular type of data. Data types determines what value can be assigned to variable and operation you can perform on it.

In this chapter we will discuss native data types in Python and PowerShell and how you can change the data types from one to another.

Common Data types

Numbers

In Python and PowerShell numbers written without decimals is considered as an integer and numbers with decimals are considered as a floating point number (or float), so `1` is an integer and `1.0` is a floating point number. These number data types can be used as arithmetic labels in a programming language and to perform basic math operations, like addition, subtraction etc.

That brings us to first number data type an Integer.

Integers

Integers in any programming language are whole numbers that can be positive or negative, including zero (`0`) and an integer in both PowerShell and Python can also be called as an '`int`'.

To declare and assign an `int` value to a Python variable just use the variable name, followed by assignment operator and the integer number, like

```
num = -12
print(-12)
```

Output:

```
-12
```

This is exactly how PowerShell integers are defined

```
$num = -12
$num
```

Output:

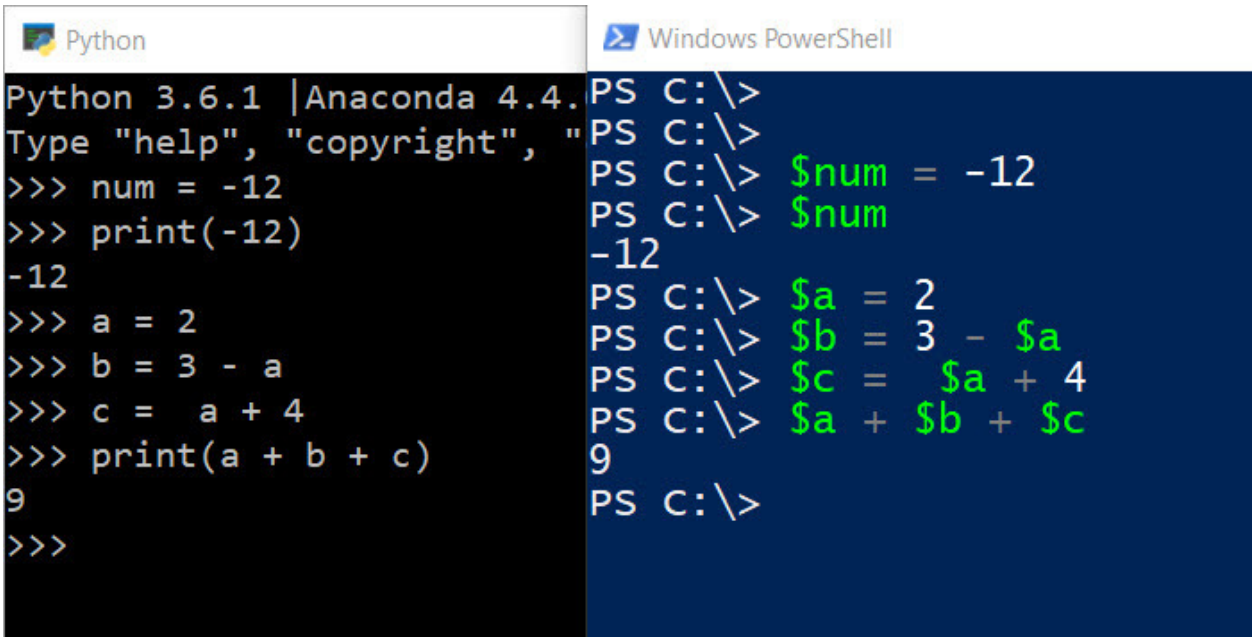
```
-12
```

We can also do basic maths with these `int` variables, like in the following Python example

```
a = 2
b = 3 - a
c =  a + 4
print(a + b + c)
```

Similarly in PowerShell to perform same basic math operations.

```
$a = 2
$b = 3 - $a
$c =  $a + 4
$a + $b + $c
```



As you continue to learn more about Python you will have a lot more opportunities to work with integers and understand about this data type.

Floating-Point Numbers

A Floating point number has a fractional part, such as `3.14` or in simpler terms any number in Python or PowerShell that contains a decimal point can be called as a Floating point number or a `float`.

Just like we did with the integer, we can create a `float` by using a decimal number.

```
num = 3.14
print(num)
```

Output:

```
3.14
```

Similarly in PowerShell to define a variable with a floating point number

```
$num =  3.14
$num
```

Output:

```
3.14
```

Booleans

Boolean data type can have one of two values, either `True` or `False` and `0` or `1`. Booleans are intended to represent the two truth values of a logic or Boolean algebra

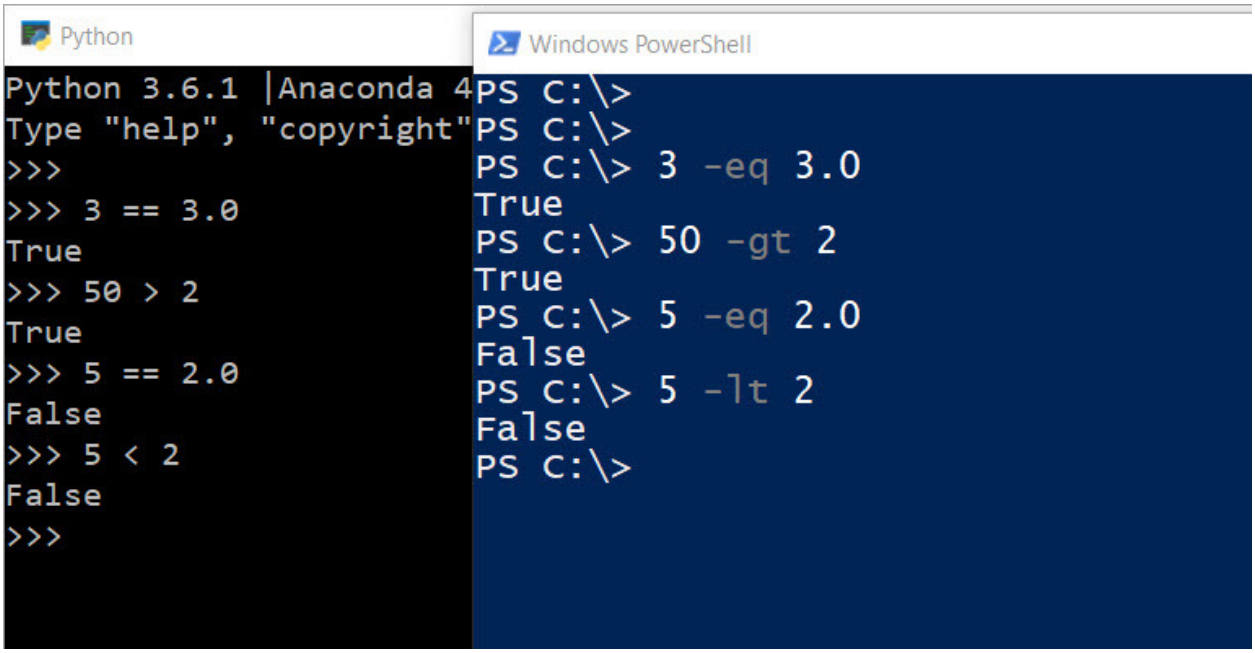
Boolean data type value `True` or `False` will always start with a capitalized T and F respectively as they are special values in Python. Whereas in PowerShell they are represented as `$true` or `$false` and is case insensitive.

Many conditional statements and operations will result in either of two values True or False, like in the following exaples:

```
3 == 3.0
True

50 > 2
True

5 == 2.0
False

5 < 2
False
```

Some of Conditional statements in PowerShell are : '`-gt`', '`-lt`' and '`-eq`', we will learn more about them in later chapters.

```
50 -gt 2
True

2 -eq 2.0
True

5 -eq 2.0
False

5 -lt 2
False
```



Strings

A sequence of one or more Alphanumeric characters is called a string, and to create a string in Python and PowerShell you've to enclose the sequence of characters in single or double quotes.

Single quotes (') are used to create strings in Python and treated the same as double quotes (''), there is no difference as far as the language syntax is concerned.

```
1  'String in single quotes'
2  "String in double quotes"
```

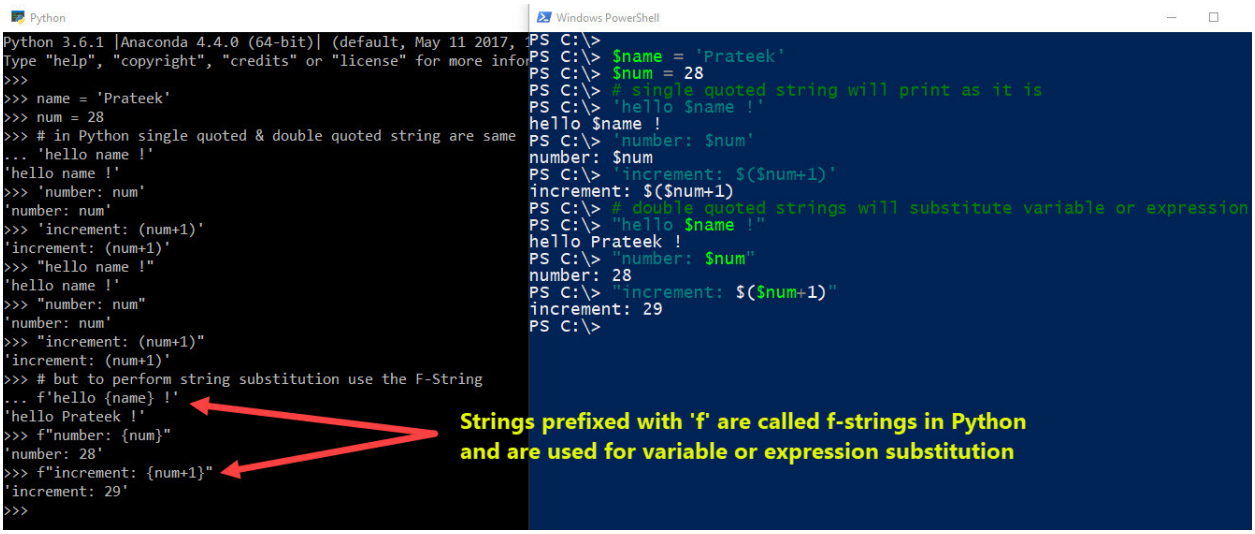
String Interpolation or Variable Substitution

In PowerShell double quoted string is used for String interpolation or variable/expression substitution, whereas single quoted strings can not perform substitution of variable, or evaluation of expression and its substitution in the string.

```powershell
$name = 'Prateek'
$num = 28
# single quoted string will print as it is
'hello $name !'
'number: $num'
'increment: $($num+1)'
# double quoted strings will substitute variable or expression
"hello $name !"
"number: $num"
"increment: $($num+1)"
```

To perform a variable substitution in Python we use a `F-String` or also known as "formatted string literals"

```python
name = 'Prateek'
num = 28
# in Python single quoted & double quoted string are same
'hello name !'
'number: num'
'increment: (num+1)'
"hello name !"
"number: num"
"increment: (num+1)"
# but to perform string substitution use the F-String
f'hello {name} !'
f"number: {num}"
f"increment: {num+1}"
```



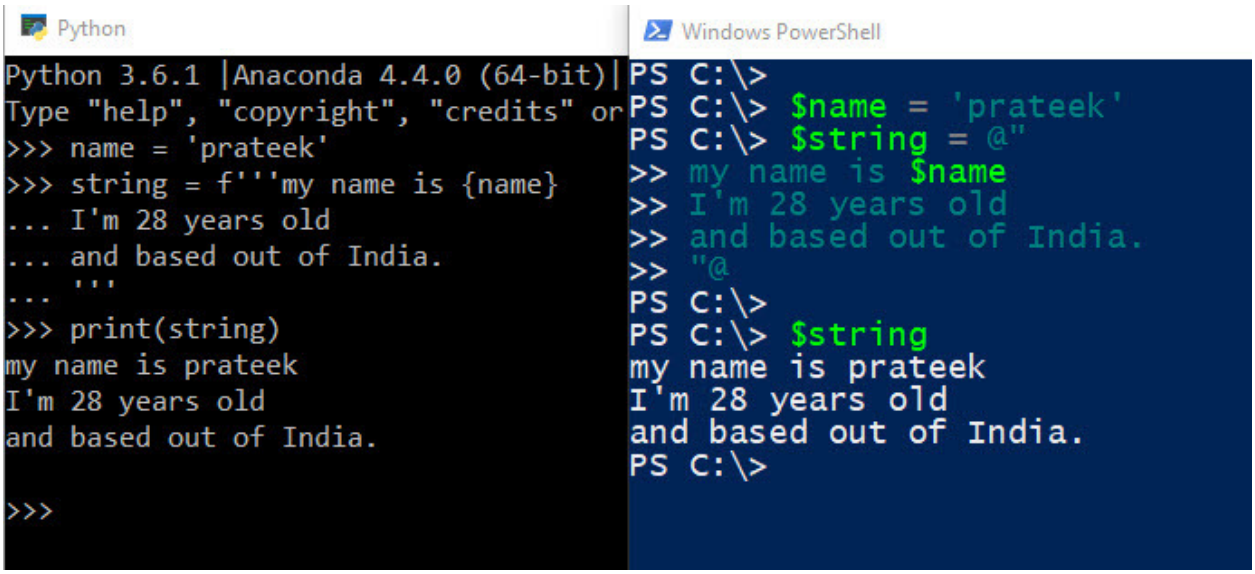
Multi-line String

To create a multi-line strings in Python we use triple quotes, this allows developer to declare strings that are not limited to just one line and can spread across multiple lines, like in the following example we defined a 3 line string with formatted string literals (`F-Strings`) for variable substitution.

```
name = 'prateek'
string = f'''my name is {name}
I'm 28 years old
and based out of India.
'''
print(string)
```

On other hand, in PowerShell you define a multi-line string just like any regular string in single or double quotes but they have a ‘ `@` ‘ on each end. Multi-line string in PowerShell is also known as `Here-Strings`.

```
$name = 'prateek'
$string = @"
my name is $name
I'm 28 years old
and based out of India.
"@
```



Just like the numbers, many operations can perform on strings within a programs to manipulate it, like concatenations, slicing, substring etc. String is an important data type because it is used to

display and communicate information between the end user and the program which makes String the most popular data type of all.

In the later chapters of this book we will learn more about string operations, manipulation and formating.

Escape Characters in String

Backslash '\' is a representation of Escape characters in a Python string and can be interpreted in a single, double or triple quoted strings. Purpose of escape character is to represent whitespace characters like: '\t' (tab), '\n' (newline), and '\r' (carriage return). An escape character can be prefixed to a special character to convert it into an ordinary character.

```
# escape characters in string
# represented by a backslash '\'
# interpreted in a single and double quoted strings.
print("\\") # backslash (\)
print('\'') # single quote (')
print("\"") # double quote (")
print("Hello\nWorld!") # linefeed (\n)
print("Hello\tWorld!")  # tab (\t)
```

Unlike Powershell which represents Escape character by backward apostrophe/grave (`) and does not interpret Escape characters in Single Quotes, but only in Double quotes.

```
# escape characters in string
# represented by a backward apostrophe/grave '`'
# interpreted in double quoted strings. NOT in single quotes
"`\" # backslash (\)
"`'" # single quote (')
"`"" # double quote (")
"Hello`nWorld!" # linefeed (\n)
"Hello`tWorld!"  # tab (\t)
```

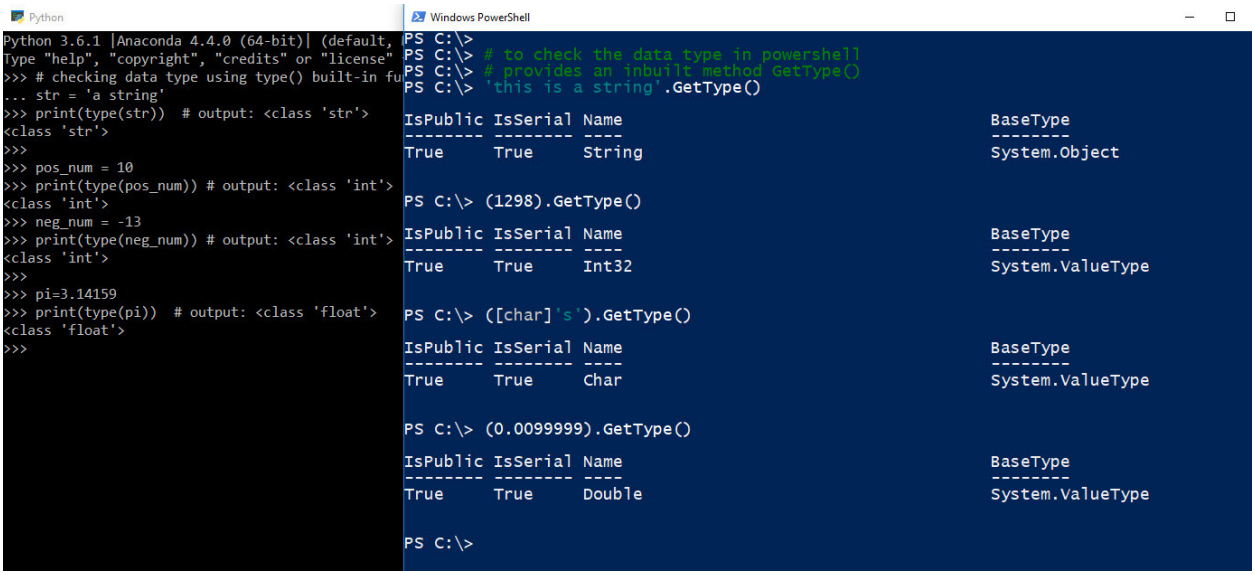
Checking the Data Type

Every Powershell Object automatically has a in-built method `GetType()`, which could be accessed using the (' . ') dot operator, like demonstrated in following example to check the data type.

```powershell
# to check the data type in powershell
# provides an inbuilt method GetType()
'this is a string'.GetType()
(1298).GetType()
([char]'s').GetType()
(0.0099999).GetType()
```

Similarly, in Python we can use the `type()` inbuilt method to test/check the data type.

```python
# checking data type using type() built-in function
str = 'a string'
print(type(str))  # output: <class 'str'>

pos_num = 10
print(type(pos_num)) # output: <class 'int'>
neg_num = -13
print(type(neg_num)) # output: <class 'int'>

pi=3.14159
print(type(pi))  # output: <class 'float'>
```



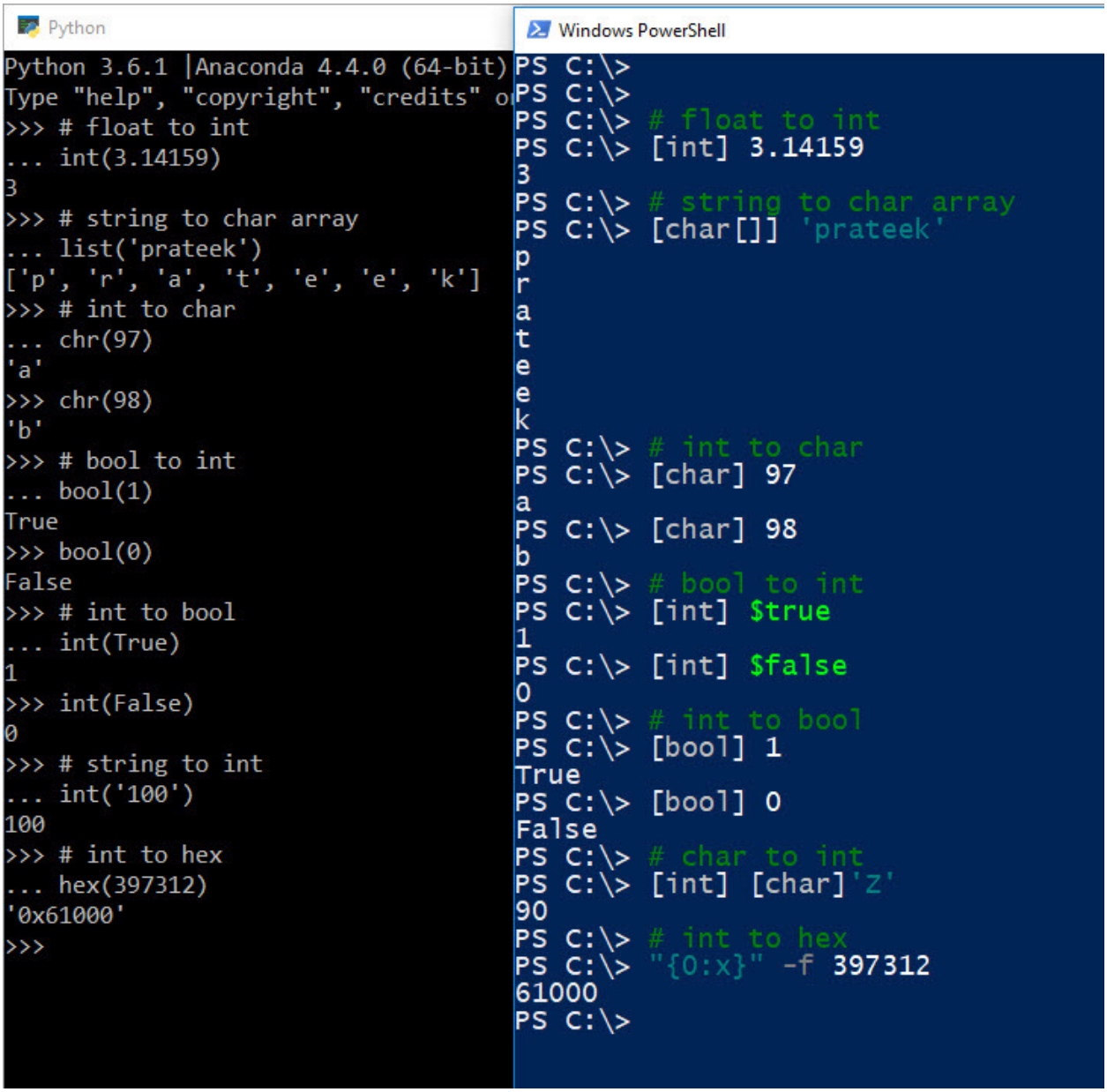
Data type casting

In any programming language it is often necessary to perform 'Type conversion' between the data types, in simpler terms changing one data type into another. Python provides some built-in functions to convert between types and most of the times you simply use the type name, like `int`,`float` as a function name `int()`, `float()`.

```
# float to int
int(3.14159)
# string to char array
list('prateek')
# int to char
chr(97)
chr(98)
# bool to int
bool(1)
bool(0)
# int to bool
int(True)
int(False)
# string to int
int('100')
# int to hex
hex(397312)
```

PowerShell provides type accelerators to typecast data types, 'Type Accelerators' work like aliases for .NET types and are intended to save some typing. To use type accelerators just use the type name inside brackets like `[int]`, `[bool]` or `[string]`

```
# float to int
[int] 3.14159
# string to char array
[char[]] 'prateek'
# int to char
[char] 97
[char] 98
# bool to int
[int] $true
[int] $false
# int to bool
[bool] 1
[bool] 0
# char to int
[int] [char]'Z'
# int to hex
"{0:x}" -f 397312
```



Following is a tabular comparison of Python's in-built function and PowerShell type-accelerators against the most popular data type conversions.

Python In-Built Function	PowerShell Type-Accelerator	Description
int()	[int]	Returns an integer object constructed from a number or string
float()	[float]	Returns a floating-point object constructed from a number or string

Python In-Built Function	PowerShell Type-Accelerator	Description
chr()	[char]	Returns string representation of character given by integer argument
str()	[string]	Returns a string version of an object
bool()	[bool]	Converts an argument to a Boolean value

Chapter 9 - String Manipulations and Formatting

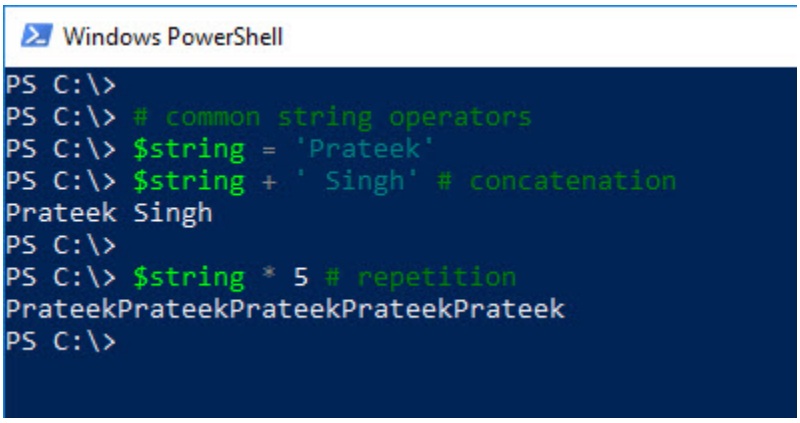
Strings are most popular data types in Python and PowerShell, that can be created by simply enclosing characters in a single or double quote. We discussed basics of Strings, String interpolation or variable substitution and escape characters in the previous chapter, but in this chapter we will take a look into how to format strings and perform various operations to manipulate string to our use.

Common String Operations

Some common string operation are Concatenation which is adding two strings together using addition operator (+) to form one string, or repetition of strings using the multiplication operator (*).

Following examples demonstrate how to perform these operations in Python

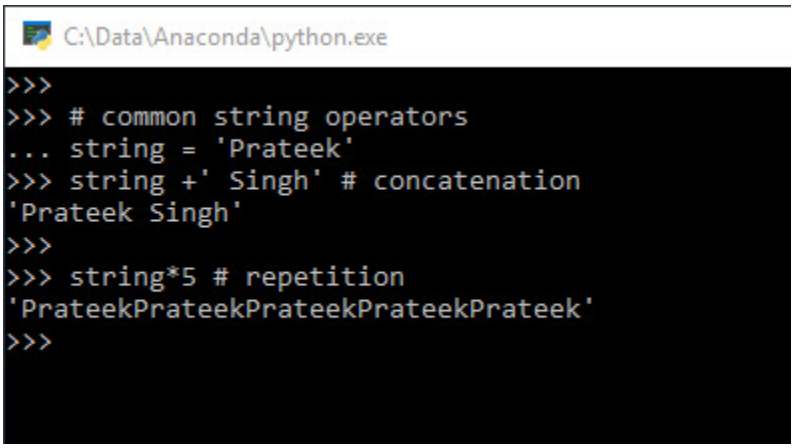
```
# common string operators
string = 'Prateek'
string +' Singh' # concatenation
string*5 # repetition
```



Common String Operations

Similarly in PowerShell:

```
# common string operators
$string = 'Prateek'
$string + ' Singh' # concatenation
$string * 5 # repetition
```



Common String Operations

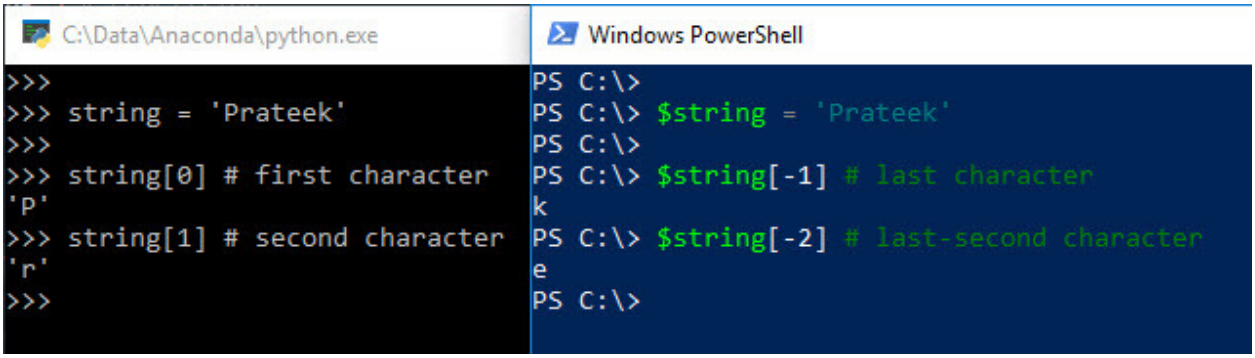
Both Python and PowerShell allows, access to a character at a specific index using brackets `[]` to define the index number of a character you want access, treating the whole String as an Array of characters.

That means to access first character of String in Python we use index '0' or `[0]` :

```
# accessing character at index in Python
string[0] # first character
string[1] # second character
```

Similarly in PowerShell index number `-1` or `[-1]` denotes the last character of a String.

```
# accessing character at index in PowerShell
$string[-1] # last character
$string[-2] # last-second character
```



Common String Operations

String Formatting

In this section we will learn various approaches to format strings in Python and PowerShell and what are the similarities between these approaches.

Format Operator in PowerShell : ' -f '

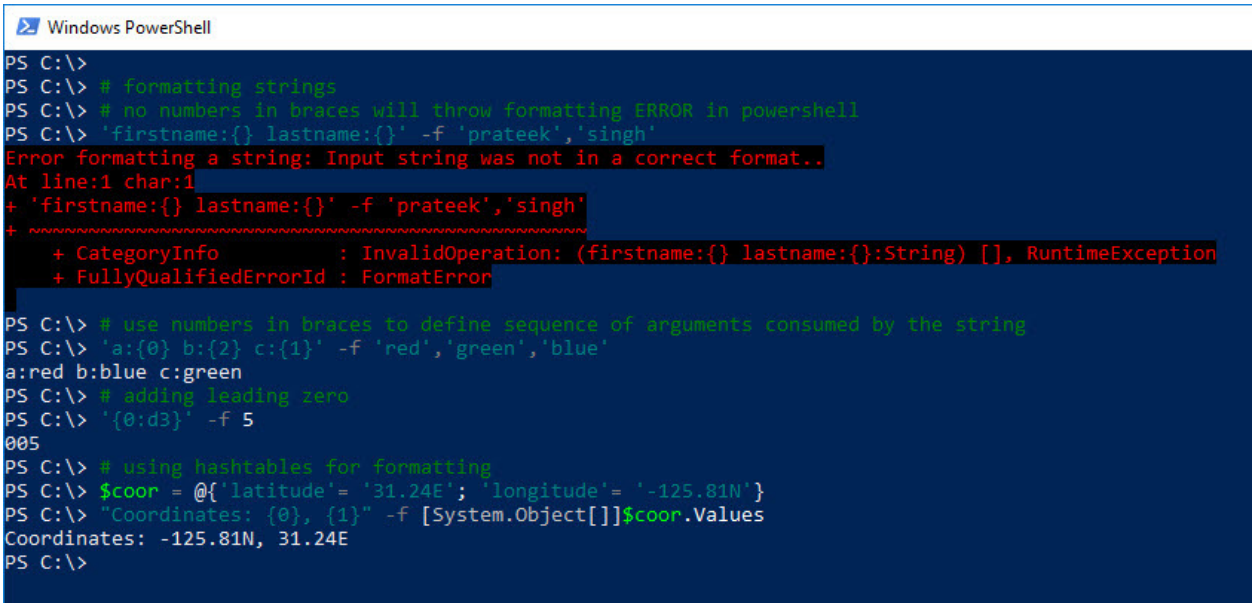
PowerShell provides a `'-f'` operator called as the Format operator to perform string formatting, where left-hand side of operator is a String with placeholders denoted by squiggly brackets (`{}`) and right-hand side is an array of values to place into the place holder string on the left.

Syntax:

```
"this is {0} a string {1}" -f $value1, $value2
```

It is a must to define each placeholder with the index of element on right hand side so that the values are placed correctly, first element of the array of values start at `'0'`

```
# formatting strings
# no numbers in braces will throw formatting ERROR in powershell
'firstname:{} lastname:{}' -f 'prateek','singh'
# use numbers in braces to define sequence of arguments consumed by the string
'a:{0} b:{2} c:{1}' -f 'red','green','blue'
# adding leading zero
'{0:d3}' -f 5
# using hashtables for formatting
$coor = @{'latitude'= '31.24E'; 'longitude'= '-125.81N'}
"Coordinates: {0}, {1}" -f [System.Object[]]$coor.Values
```



String Formatting in PowerShell

Format Method in Python : ‘ .format() ‘

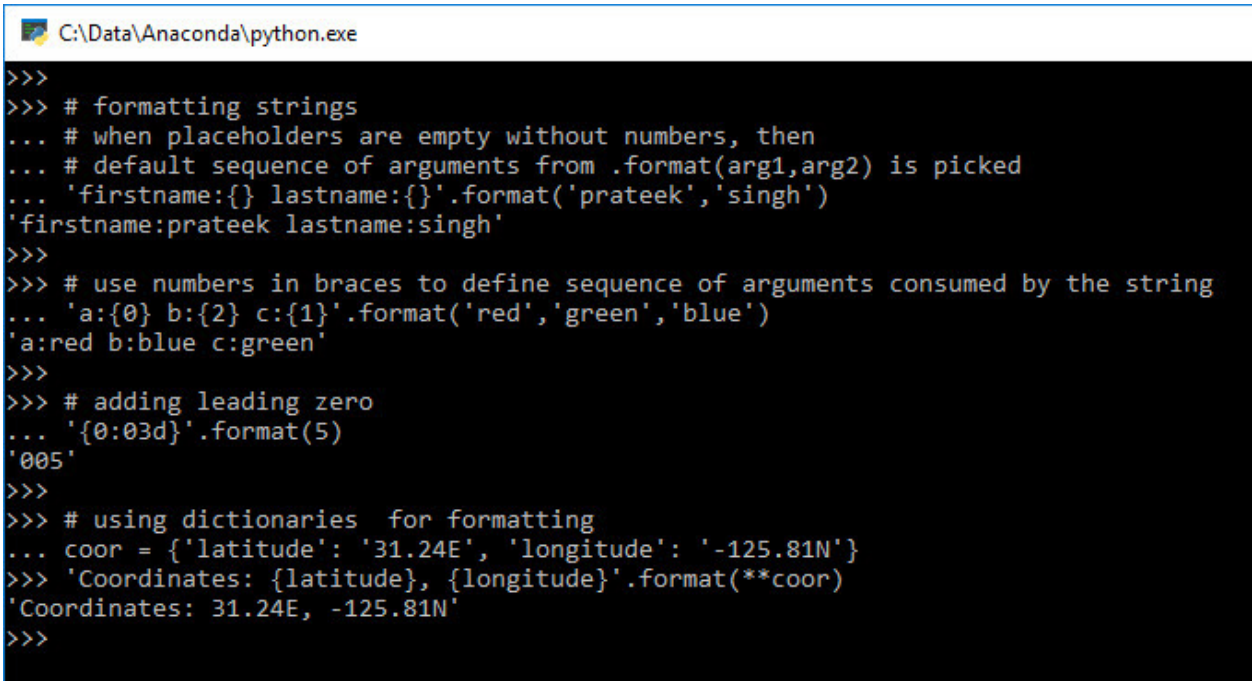
Python has a similar Built-in `.format()` method for formatting the strings, which can be applied on any string on the left using the Dot (.) operator and accepts values as arguments that will passed into the placeholders in the string.

Syntax:

```
"this is {} a string {}".format(value1, value2)

Alternatively with index numbers

"this is {0} a string {1}".format(value1, value2)
```

It is not mandatory to define the index inside a placeholder and Python automatically assumes the placeholder value in the sequence they are defined in `format()` method as a argument.

```
# formatting strings
# when placeholders are empty without numbers, then
# default sequence of arguments from .format(arg1,arg2) is picked
'firstname:{} lastname:{}'.format('prateek','singh')
# use numbers in braces to define sequence of arguments consumed by the string
'a:{0} b:{2} c:{1}'.format('red','green','blue')
# adding leading zero
'{0:03d}'.format(5)
# using dictionaries  for formatting
coor = {'latitude': '31.24E', 'longitude': '-125.81N'}
'Coordinates: {latitude}, {longitude}'.format(**coor)
```



String Formatting in Python

Substring or String Slicing

PowerShell and Python, both treat Strings as an array of characters, so we can use an array's functionality called "slicing" on our strings to substring a string, which slices a piece of string from the original string.

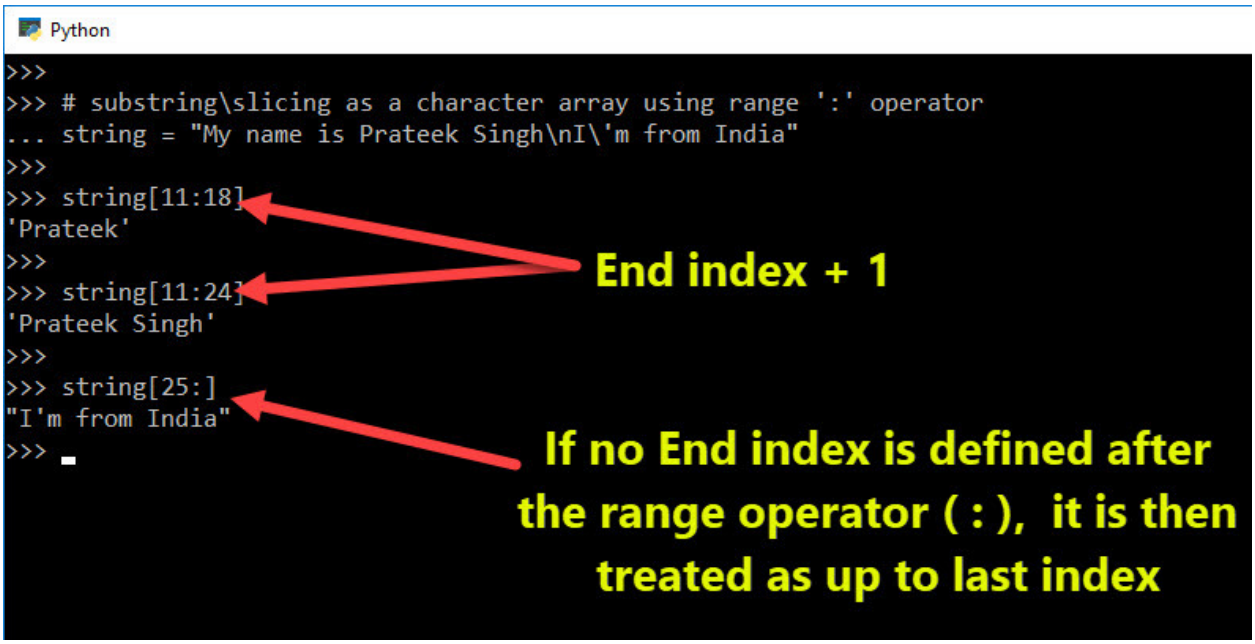
Following are couple of ways to perform a substring of a string:

Using Range operator

```powershell
# substring\slicing as a character array using range '..' operator
$string = "My name is Prateek Singh`nI`'m from India"

# slicing the string from a specific index to another
$string[11..17] -join ''  # output: Prateek
$string[11..23] -join ''  # output: Prateek Singh

# slicing up to last index
$string[25..($string.Length-1)] -join '' # output: I'm from India
```

String Slicing using Range Operator in PowerShell

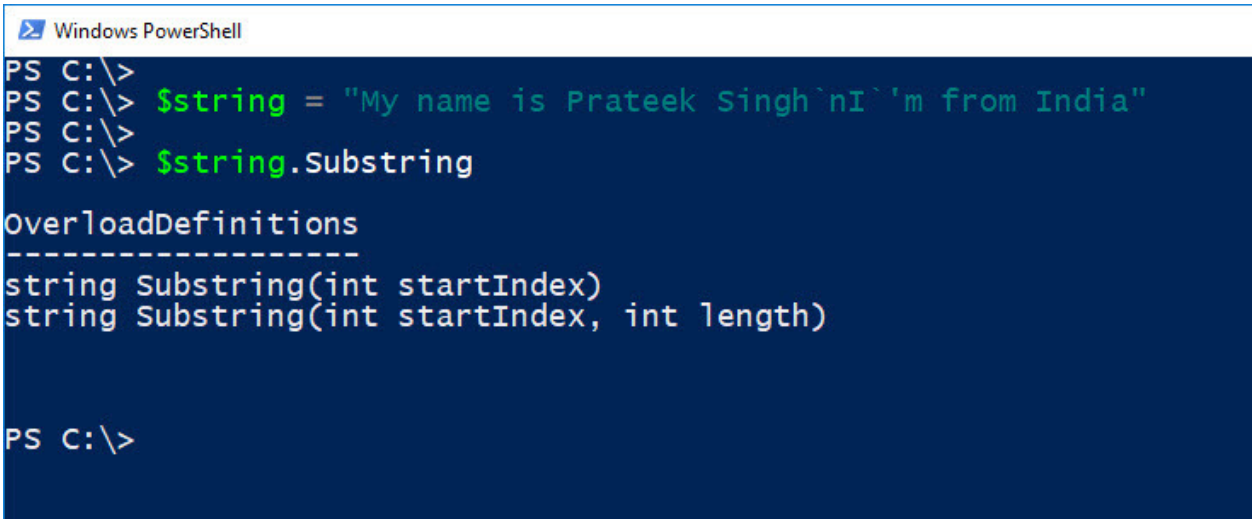
```python
# substring\slicing as a character array using range ':' operator
string = "My name is Prateek Singh\nI\'m from India"

# slicing the string from a specific index to another
string[11:18] # output: Prateek
string[11:24] # output: Prateek Singh

# slicing up to last index
string[25:] # output: I'm f rom India
```



String Slicing using Range Operator in Python

Using Substring() method

PowerShell on other hand also offers a `SubString()` method inbuilt into every string object which accepts starting index and length of substring like in the following examples:



Method Overload definitions of .SubString()

```
# substring using .substring() method
$string = "My name is Prateek Singh`nI`'m from India"
$string.Substring(11,7) # output: 'Prateek'
$string.Substring(25,14)  # output: 'I'm from India'
$string.Substring(11,13)  # output: 'Prateek Singh'
```

String Slicing using SubString() method

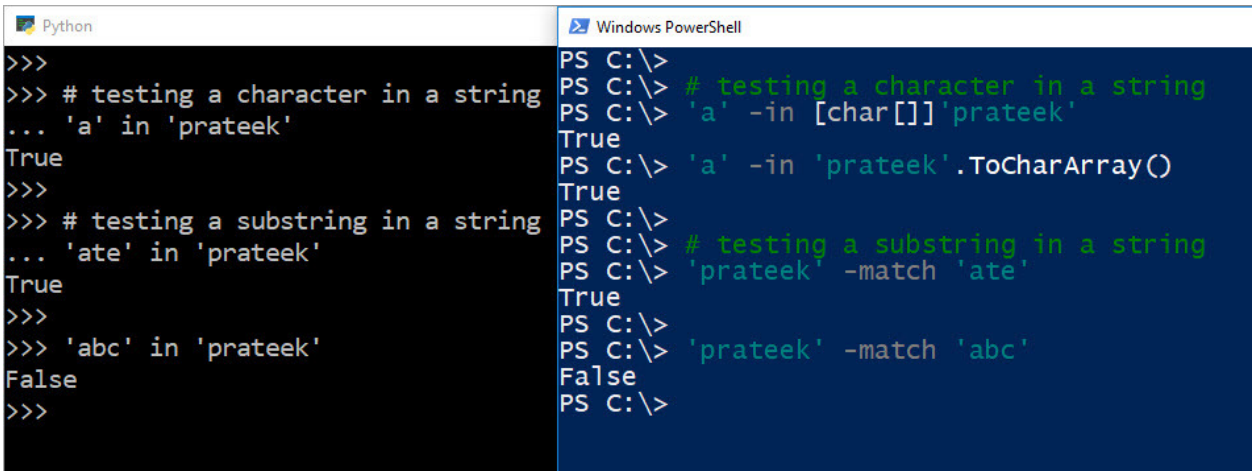
Testing String Membership

We can test if a character or a substring exists within a string or not. In Python we use the keywords like `'in'` or `'not in'`:

```python
# testing a character in a string
'a' in 'prateek'

# testing a substring in a string
'ate' in 'prateek'
'abc' not in 'prateek'
```

Whereas, in PowerShell we use the operators such as `'-in'` for characters and `'-match'` and `'-notmatch'` for testing substring memberships:

```powershell
# testing a character in a string
'a' -in [char[]]'prateek'
'a' -in 'prateek'.ToCharArray()

# testing a substring in a string
'prateek' -match 'ate'
'prateek' -notmatch 'abc'
```



Built-in methods

Python and PowerShell provide built-in methods for string manipulations, which are up to some extent similar in functionalities and very useful in order to perform robust string manipulations in programs or in interactive use cases.

Following examples demonstrate some common use cases of String Manipulations with built-in methods in PowerShell and Python:

Length of string and Finding Character at an Index

You can find length of a string object by accessing it's `'length'` property and to find starting index of a specific character or substring use `IndexOf()` method in PowerShell.

```
# built-in string methods
'powershell'.Length # length of string
'powershell'.IndexOf('s') # find characters at an index
```

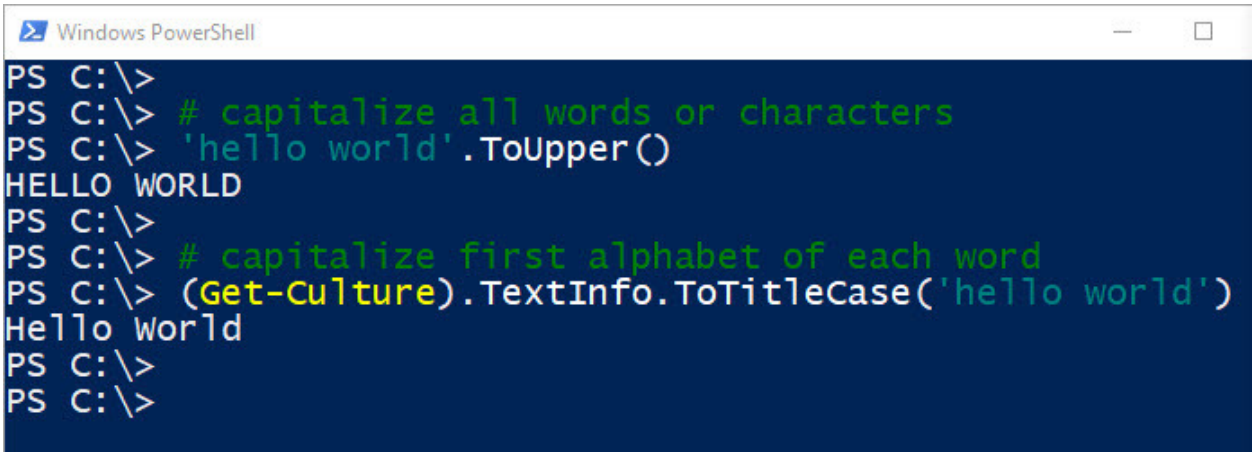
Python provides an inbuilt method called `index()` that will return the starting index of a character or a substring.

```
# built-in string methods
len('python') # length of string
'python'.index('o') # find characters at an index
```

Changing the Case

To change the case of string in PowerShell use the '`ToUpper()`' this will change the every character of the string to upper case, and in case you want to only capitalize the first alphabet of each word in the string then use `ToTitleCase()` method like in the following example:

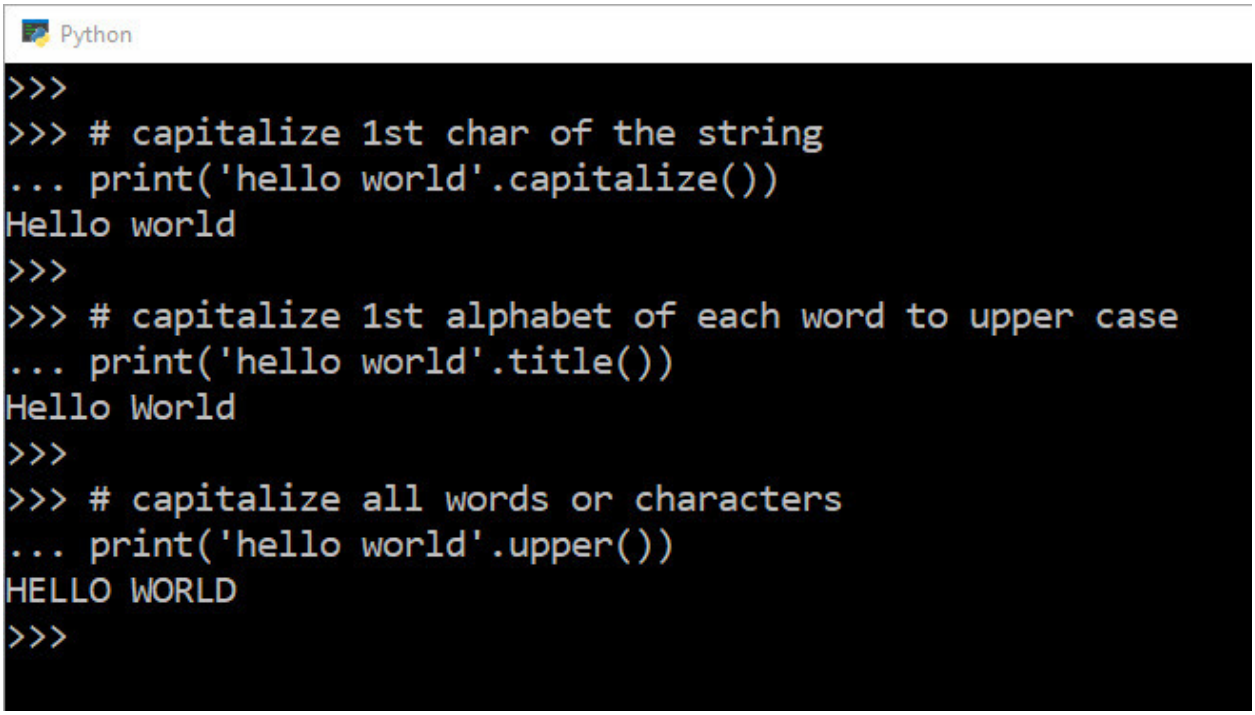
```
# capitalize all words or characters
'hello world'.ToUpper()
# capitalize first alphabet of each word
(Get-Culture).TextInfo.ToTitleCase('hello world')
```



Changing Case of a String in PowerShell

Python has `capitalize()` inbuilt method to change the first char of a string to uppercase, and `title()` method to capitalize first alphabet of every word. To modify each and every character in the string to uppercase one should use the `upper()` method.

```
# capitalize 1st char of the string
print('hello world'.capitalize())
# capitalize 1st alphabet of each word to upper case
print('hello world'.title())
# capitalize all words or characters
print('hello world'.upper())
```



Changing Case of a String in PowerShell

Split and Join Strings

While manipulating strings you will come across scenarios where you require to spilt a string into two or more parts by a character or word, and even join two or more characters\strings to form one string. Following are examples to demonstrate how to split and join strings in PowerShell using: `-Split`, `split()` and `-Join`

```powershell
# splitting a string .split() -split
$string = "My name is Prateek Singh`nI`'m from India"
$string.split(' ')  # splitting the string by white space
$string -split 'prateek'  # splitting the string by a word, eq. 'my'
$string -split '\n'  # splitting by newline character

# joining string .join() -join
[char[]]'Prateek' -join ' '  # Joining each char with a space
[char[]]'Prateek' -join '*'  # Joining each char with a '*'
```

Splitting and Joining strings in PowerShell

Python also offer a similar functionality using the `.split()` and `.join()` inbuilt methods.

```python
# splitting a string .split()
print(string.split(' '))  # splitting the string by white space
print(string.split('is'))  # splitting the string by a word, eq. 'is'
print(string.split('\n'))  # splitting by newline character

# joining string .join()
print(' '.join('Prateek'))  # Joining each char with a space
print('*'.join('Prateek'))  # Joining each char with a '*'
```

```
Python
>>>
>>> # splitting a string .split()
... print(string.split(' '))  # splitting the string by white space
['My', 'name', 'is', 'Prateek', "Singh\nI'm", 'from', 'India']
>>>
>>> print(string.split('is'))  # splitting the string by a word, eq. 'is'
['My name ', " Prateek Singh\nI'm from India"]
>>>
>>> print(string.split('\n'))  # splitting by newline character
['My name is Prateek Singh', "I'm from India"]
>>>
>>> # joining string .join()
... print(' '.join('Prateek'))  # Joining each char with a space
P r a t e e k
>>>
>>> print('*'.join('Prateek'))  # Joining each char with a '*'
P*r*a*t*e*e*k
>>>
```

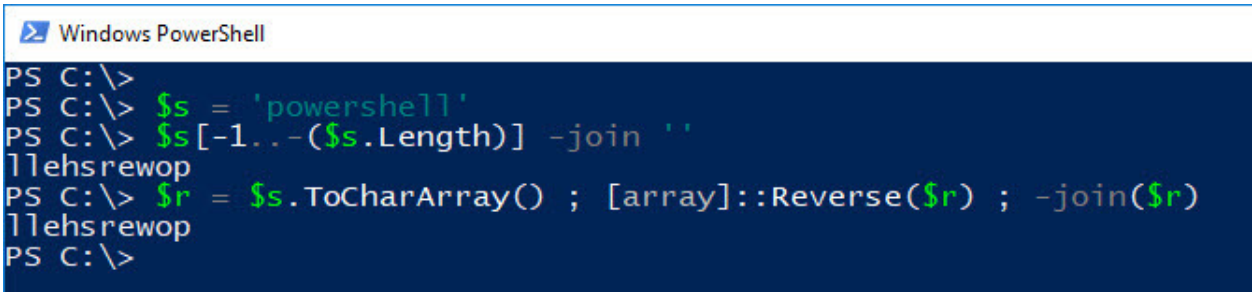
Splitting and Joining strings in PowerShell

Reversing a string

To reverse a string in PowerShell, you have to access each character of the string from last index: `-1` to the index: negative of 'length of string' and then perform a join operation to concatenate all characters to form a reversed string.

There is also an alternative: which is to convert string object to character array using `ToCharArray()` method and then use Array type accelerator's `[array]` method named `Reverse()` to reverse the sequence of characters, finally `-join()` characters to form a reversed string.

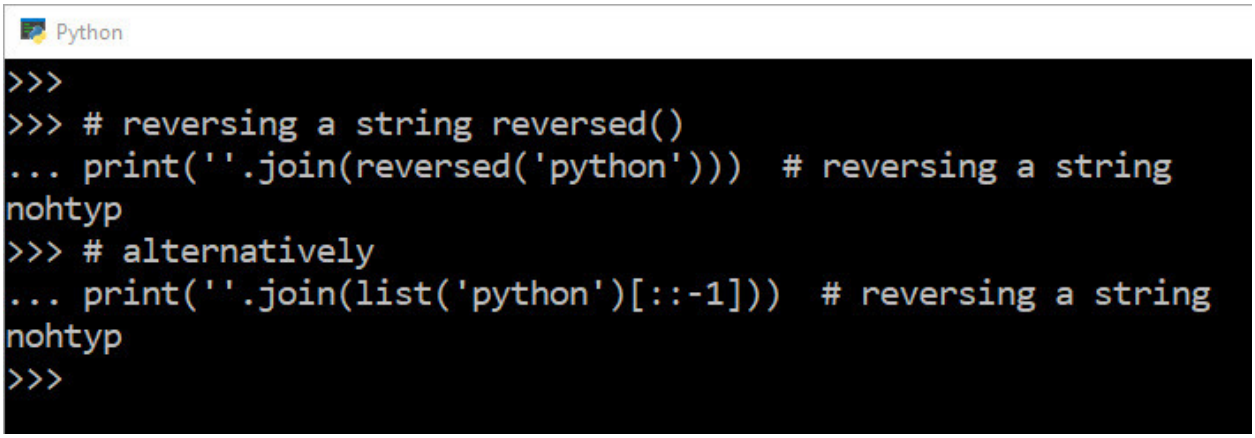
```
1  $s = 'powershell'
2  $s[-1..-($s.Length)] -join ''
3  $r = $s.ToCharArray() ; [array]::Reverse($r) ; -join($r)
```



Reversing String in PoweShell

Whereas in Python, it is fairly simple to reverse the strings using `reversed()` in built method, that will return character sequence in reverse and you just need to join these characters using `''.join()` method like in the following example to form the reversed string:

```
1  # reversing a string reversed()
2  print(''.join(reversed('python')))  # reversing a string
3  # alternatively
4  print(''.join(list('python')[::-1]))  # reversing a string
```

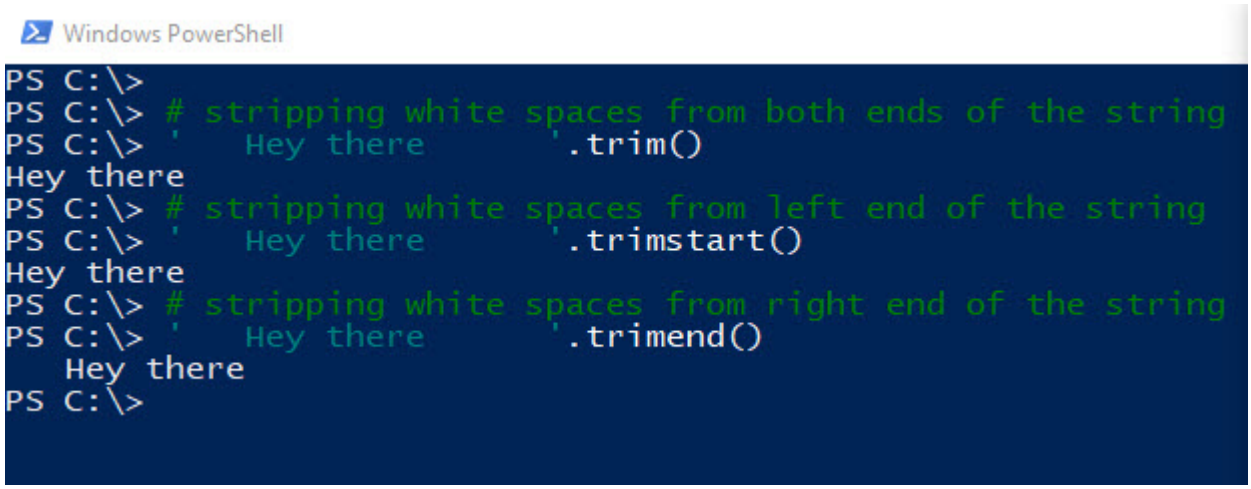


Reversing String in Python

Removing Whitespace

PowerShell has `'trim*()'` methods to perform removal of any white spaces from left, right or both side of the string, these are helpful in various scenarios where white spaces are introduced during string manipulations and are not required.

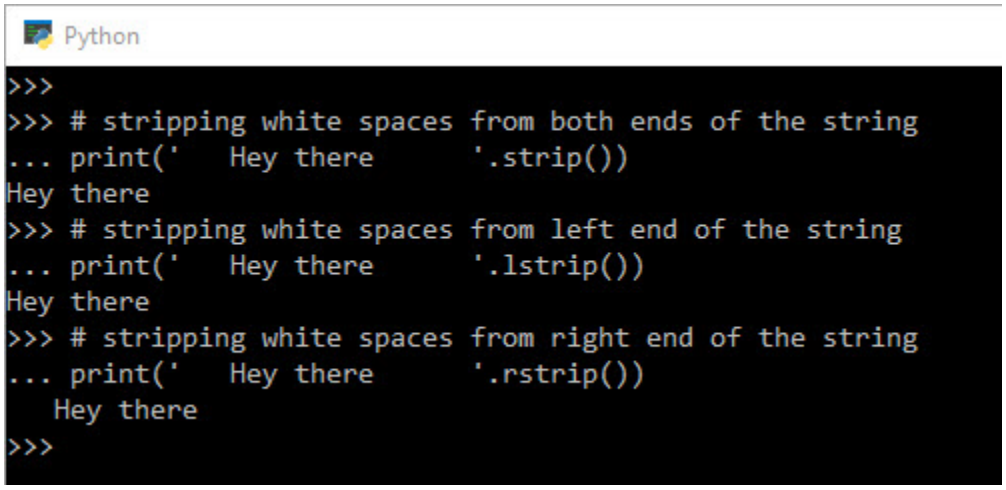
```
# stripping white spaces from both ends of the string
'   Hey there      '.trim()
# stripping white spaces from left end of the string
'   Hey there      '.trimstart()
# stripping white spaces from right end of the string
'   Hey there      '.trimend()
```



Removing White Spaces in PowerShell and Python

Python variants of these inbuilt methods are called: `strip()`, `lstrip()` and `rstrip()`

```
# stripping white spaces from both ends of the string
print('   Hey there      '.strip())
# stripping white spaces from left end of the string
print('   Hey there      '.lstrip())
# stripping white spaces from right end of the string
print('   Hey there      '.rstrip())
```

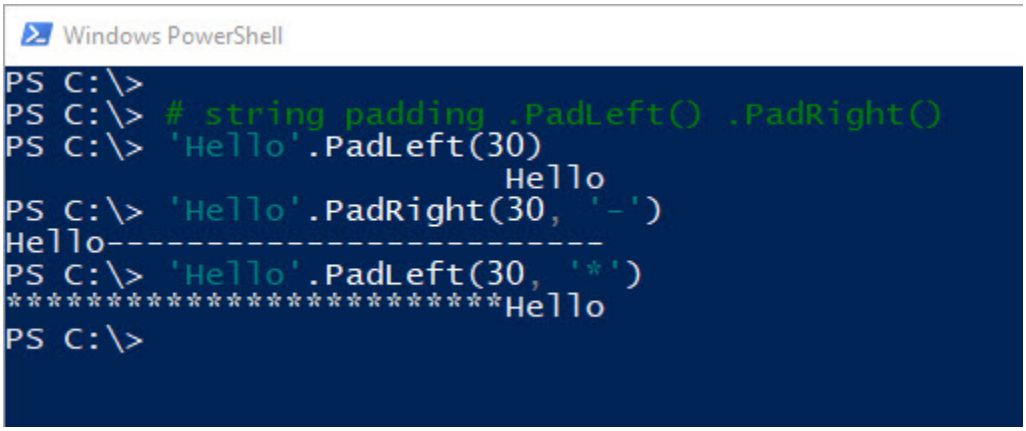


Removing White Spaces in PowerShell and Python

Text Alignment

Often when printing strings on the console you'll require to adjust text alignments, in terms of PowerShell it is called '`Padding`', so we pad strings to either left or right like in the following examples:

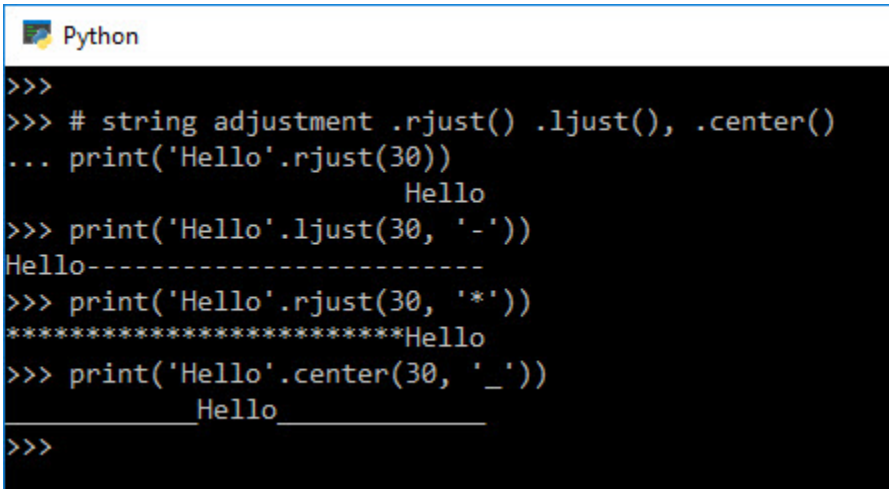
```
# string padding .PadLeft() .PadRight()
'Hello'.PadLeft(30)
'Hello'.PadRight(30, '-')
'Hello'.PadLeft(30, '*')
```



Padding String in PowerShell

Similarly in Python, we adjust text alignment to left, right or center using the following inbuilt methods:

```
# string adjustment .rjust() .ljust(), .center()
print('Hello'.rjust(30))
print('Hello'.ljust(30, '-'))
print('Hello'.rjust(30, '*'))
print('Hello'.center(30, '_'))
```



Adjusting String Alignment in Python

Chapter 10 - Date and Time

In this chapter we are going to discuss several tools PowerShell and Python provide to work with date and time, and some common scenarios and use cases you will come across when working with date and time.

Get Date and Time

Let's begin with understanding how to retrieve date and time information.

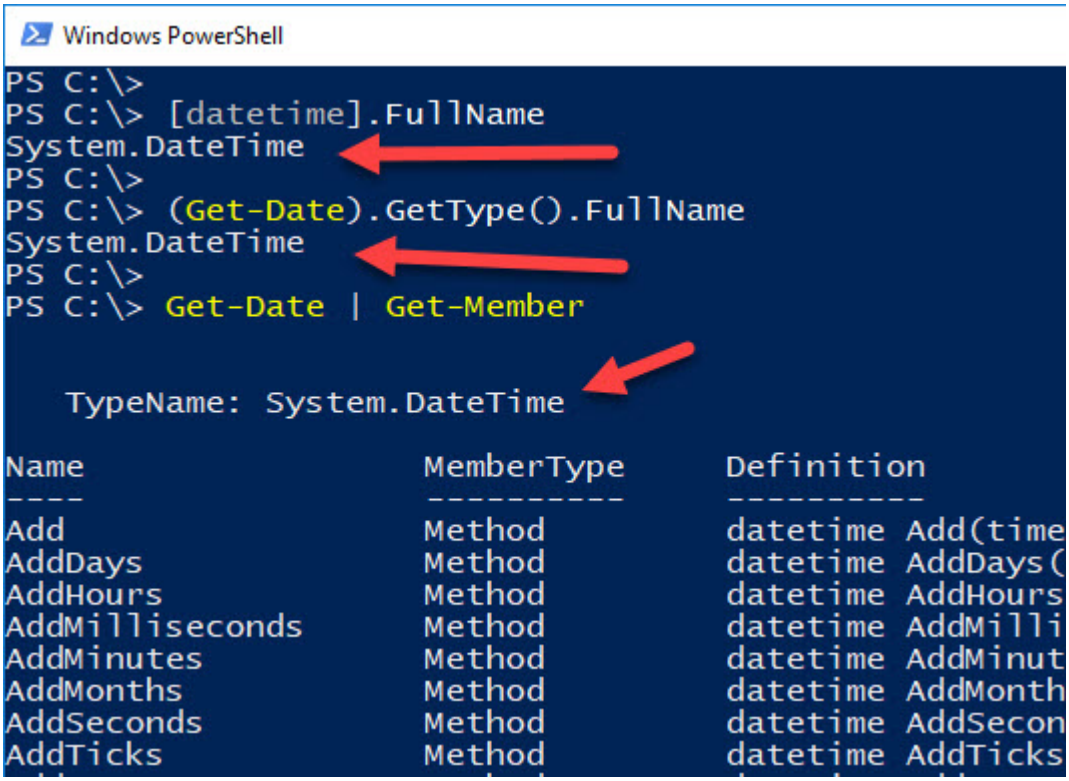
System.DateTime

Getting date and time in PowerShell is as simple a cmdlet `Get-Date` or using the `[datetime]` type accelerator.

```
# date and time
Get-date
[datetime]::Now | Format-List
[datetime]::Now # local time
[datetime]::UtcNow # time in UTC
```

Both of them under the hoods call .net class `System.DateTime` to return rich `DateTime` objects, but if you dig a little deep you'll understand, that there is no difference at all in the returned objects.

```
[datetime].FullName
(Get-Date).GetType().FullName
```



Both return objects of `DateTime` .Net class

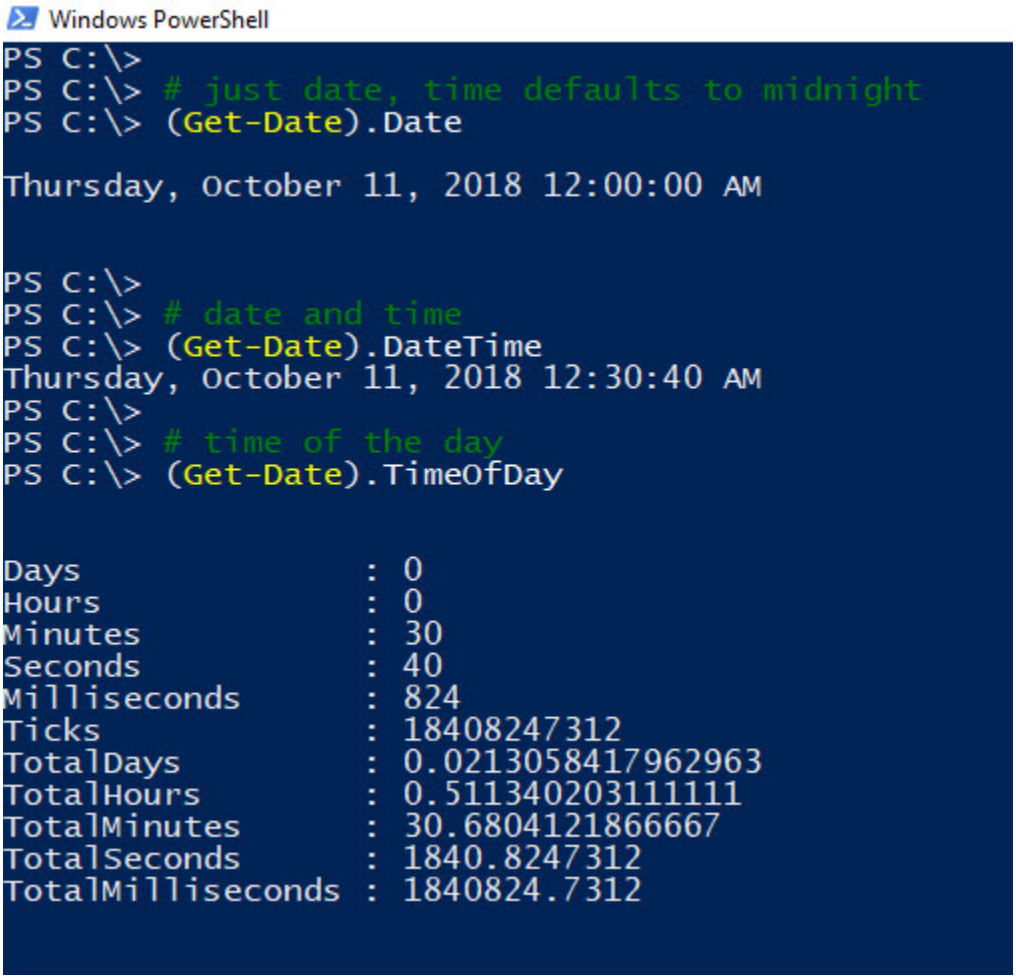
If you pipe the results of the `Get-Date` cmdlet to `Get-Member`, you will get the member properties and methods of the `DateTime` object like `Day`, `Hour`, `Minute`, `Second`, `AddDays()` and `AddHours()`.

```
Get-Date | Get-Member
```

These properties can be utilized to get current date and time:

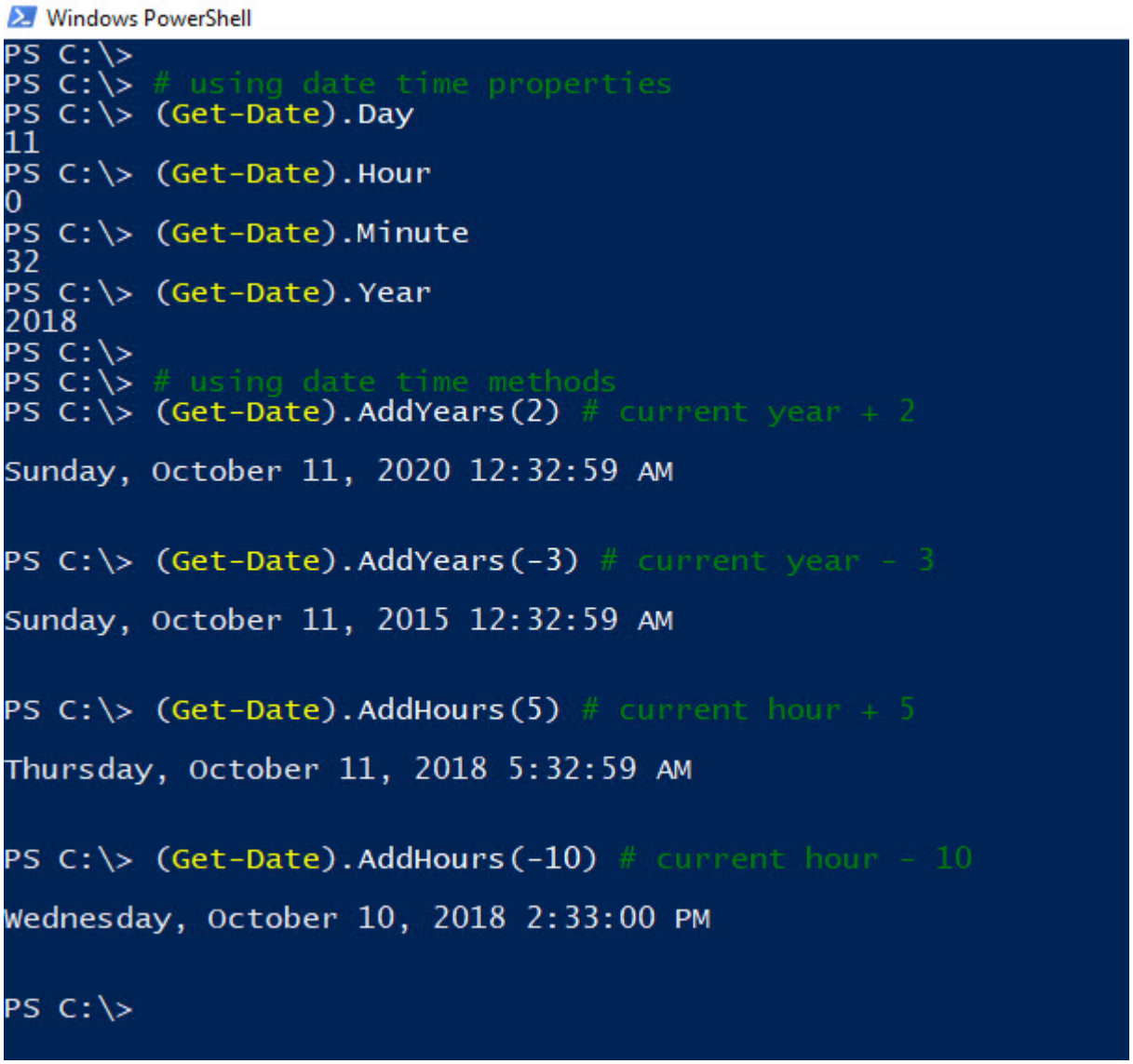
```
# using date time properties

# just date, time defaults to midnight
(Get-Date).Date

# date and time
(Get-Date).DateTime

# time of the day
(Get-Date).TimeOfDay

# other properties
(Get-Date).Day
(Get-Date).Hour
```

```
15 (Get-Date).Minute
16 (Get-Date).Year
```



Methods of class `System.DateTime` time objects can be used to perform various date time operations like adding years, hours, minutes to current date.

```
1 # using date time methods
2 (Get-Date).AddYears(2) # current year + 2
3 (Get-Date).AddYears(-3) # current year - 3
4 (Get-Date).AddHours(5) # current hour + 5
5 (Get-Date).AddHours(-10) # current hour - 10
```



On other hand, things on Python side are also simple but little confusing, so please stay with me while I explain this to you. Python provides two modules to handle Date and Time, which could be imported to your session to get date and time objects.

1. `'time'` module
2. `'datetime'` module

Alright, let's begin with the first module ...

'time' Module

Both these modules perform almost similar functionalities, causing an overlap which is the area of confusion, that makes the design rationale of two modules unclear.

The difference is that: the `'time'` module is written in low-level `'C'` language and provides low-level operating system level functionalities. Whereas, the `datetime` module is written in Python which is a high level language, historically speaking initial versions of these two modules weren't overlapping, but the overlap was developed with time as both modules evolved independently.

Following code sample demonstrates use of the `time` module.

```
# first thing is to import the time module
import time

# returns float number, UNIX representation of time
time.time() # time() -> floating point number

# returns a time tuple
time.localtime() # local time
time.gmtime(time.time()) # time in UTC

# converting time tuple to string
time.asctime(time.localtime()) # asctime([tuple]) -> string
time.asctime(time.gmtime(time.time())) # time in UTC as string
```

Please note when we are using the `time` module then time is returned in either of two standard representations of time.

1. Number of seconds since Epoch (generally January 1st, 1970) as a floating point number (to represent fractions of seconds).
2. A Tuple of 9 integers giving the local time, which includes following items:
   * Year
   * Month
   * Day
   * Hours
   * Minutes
   * Seconds
   * Weekday
   * Julian day[6]
   * DST (Daylight Saving Time)

The `'time'` module offers few more functions which are listed in the following table:

[6] https://en.wikipedia.org/wiki/Julian_day

Name	Description
`time()`	Returns current time in seconds since the Epoch as a floating point numbers
`clock()`	Returns CPU time since process start as a float
`sleep()`	Delay for a number of seconds passed to the function
`gmtime()`	Convert seconds since Epoch to UTC time tuple
`localtime()`	Converts seconds since Epoch to local time tuple
`asctime()`	Converts time tuple to a string
`ctime()`	Converts time in seconds to string
`mktime()`	Converts local time tuple to seconds since Epoch
`strftime()`	Converts time tuple to string according to a specific format
`strptime()`	Parses string to time tuple according to format specification
`tzset()`	To change the local timezone

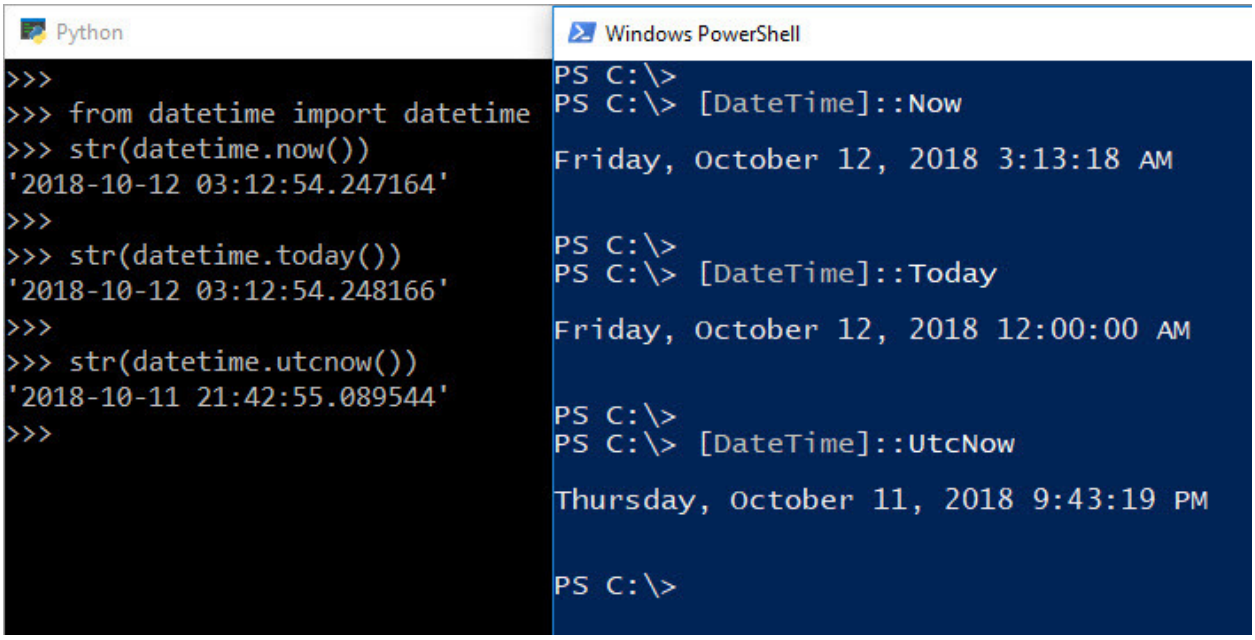
'datetime' Module

The `'datetime'` module provides simple methods for manipulating dates and times, like to get current time use `now()` method.

```
# date and time
from datetime import datetime
str(datetime.now())
str(datetime.today())
str(datetime.utcnow())
```

PowerShell has almost similar implementation, like in the following example.

```
[DateTime]::Now
[DateTime]::Today
[DateTime]::UtcNow
```

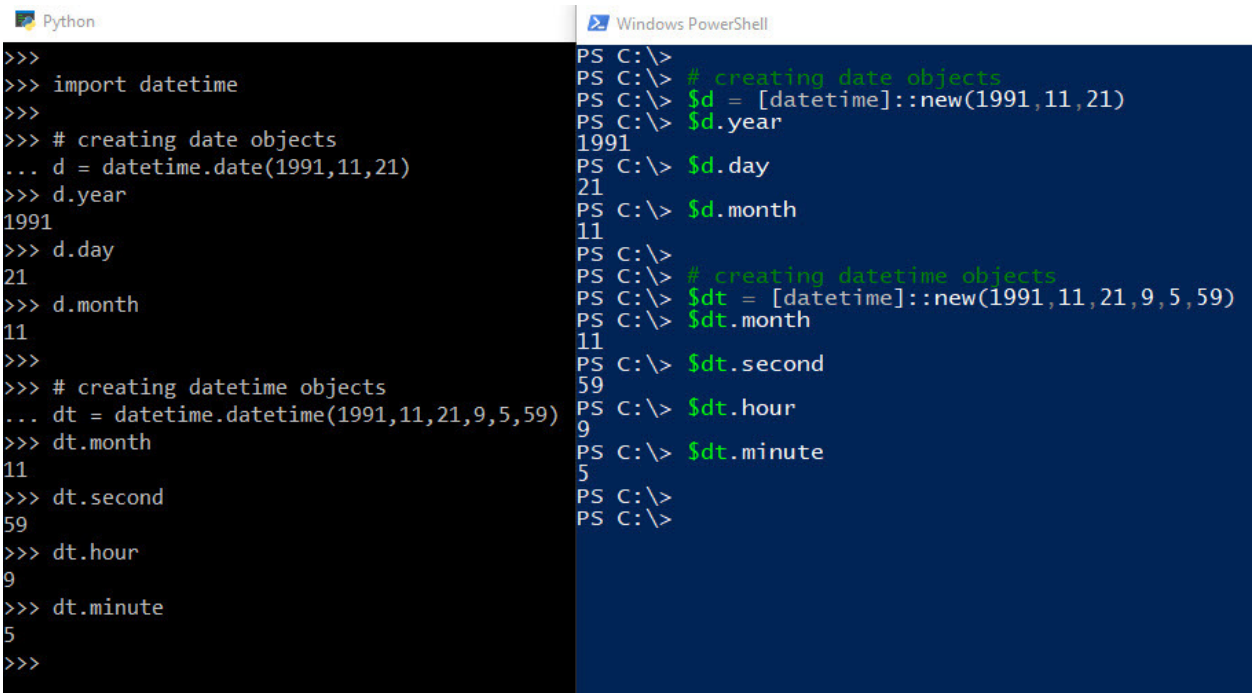


You can even use this module to create `datetime` objects in Python by providing year, month, day, hour, minute, seconds as arguments.

```
import datetime

# creating date objects
d = datetime.date(1991,11,21)
d.year
d.day
d.month

# creating datetime objects
dt = datetime.datetime(1991,11,21,9,5,59)
dt.month
dt.second
dt.hour
dt.minute
```

Similarly, PowerShell's `[DateTime]` type accelerators can be used to create date time objects and just like Python you can pass arguments to the `new()` method to create a new object.

```powershell
# to check method overload definitions,
# call the method without parenthesis
[datetime]::new

# creating date objects
$d = [datetime]::new(1991,11,21)
$d.year
$d.day
$d.month

# creating datetime objects
$dt = [datetime]::new(1991,11,21,9,5,59)
$dt.month
$dt.second
$dt.hour
$dt.minute
```



creating datetime objects

Convert String to datetime object

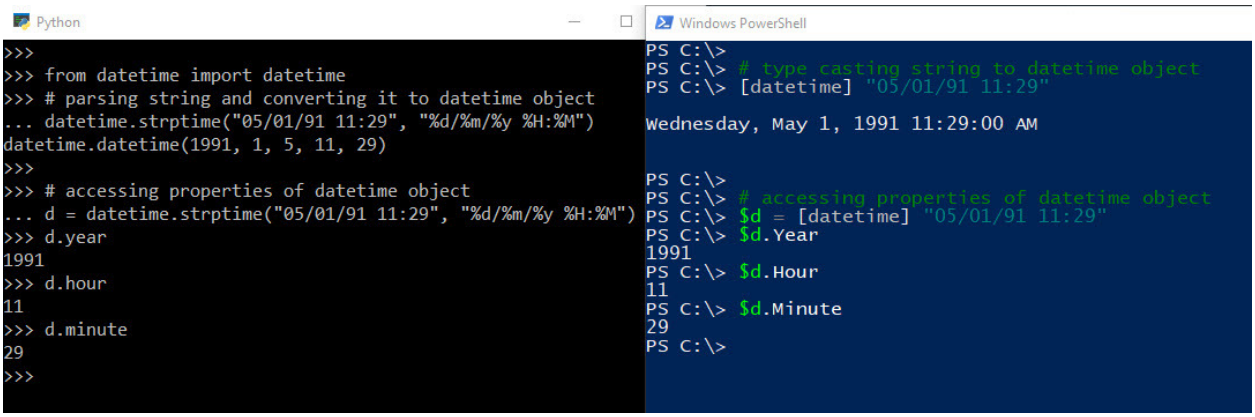
Often you'll come across scenarios where it will be required to convert a date or time in a string to datetime objects, so that it can manipulated easily.

Following examples demonstrates how to use `.strptime()` method of `datetime` module to parse strings and convert to datetime objects.

```
from datetime import datetime
# parsing string and converting it to datetime object
datetime.strptime("05/01/91 11:29", "%d/%m/%y %H:%M")

# accessing properties of datetime object
d = datetime.strptime("05/01/91 11:29", "%d/%m/%y %H:%M")
d.year
d.hour
d.minute
```

On other hand in PowerShell it is way simpler and strings can be directly type-casted to objects by defining `[datetime]` type accelerator before the string, like in the following examples.

```
# type casting string to datetime object
[datetime] "05/01/91 11:29"

# accessing properties of datetime object
$d = [datetime] "05/01/91 11:29"
$d.Year
$d.Hour
$d.Minute
```



Parsing Strings DateTime objects

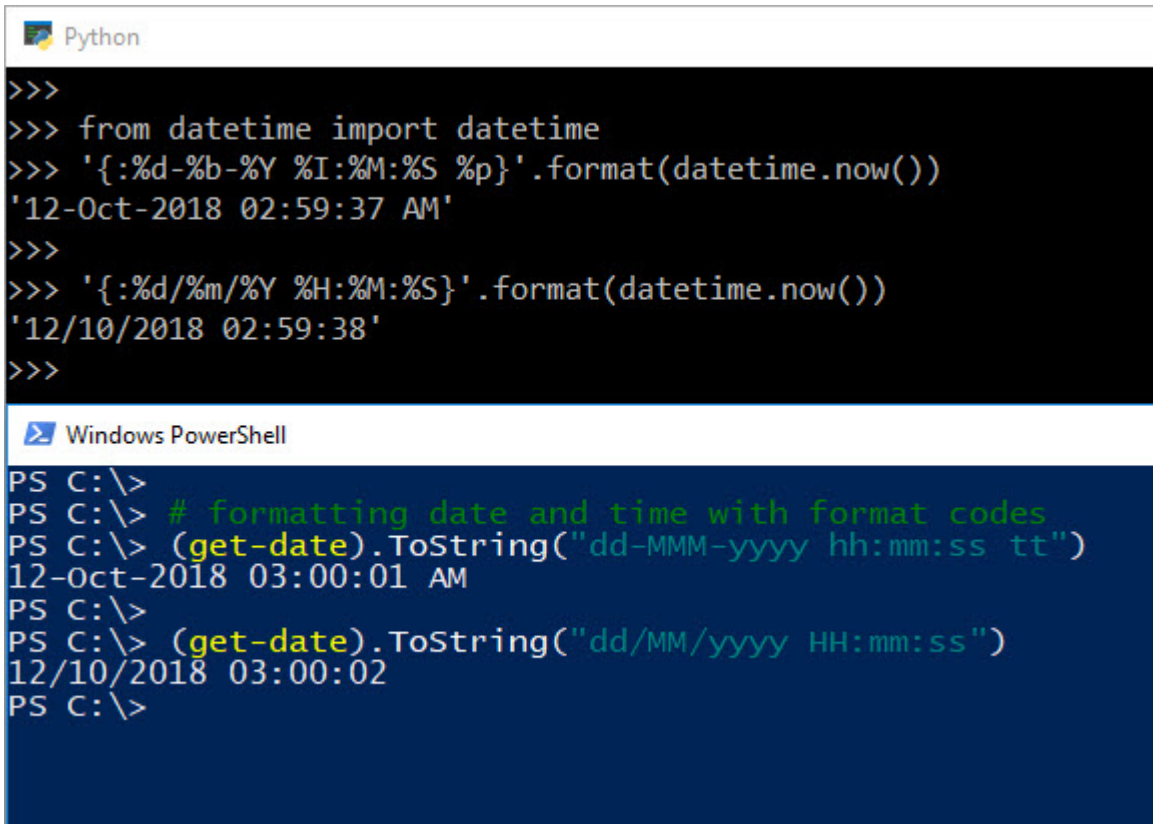
DateTime formatting

Both PowerShell and Python provides Date and Time formatting using some formatting codes which can be used to achieve desirable date-time formats. Following examples will demonstrate how to format datetime in Python using the `.format()` method and the formatting codes:

```python
# import the module
from datetime import datetime

#
'{:%d-%b-%Y %I:%M:%S %p}'.format(datetime.now())

'{:%d/%m/%Y %H:%M:%S}'.format(datetime.now())
```

Whereas, in PowerShell we can use `.ToString()` method on `System.DateTime` objects and pass the formatting codes in the string argument.

```powershell
# formatting date and time with format codes
(get-date).ToString("dd-MMM-yyyy hh:mm:ss tt")

(get-date).ToString("dd/MM/yyyy HH:mm:ss")
```



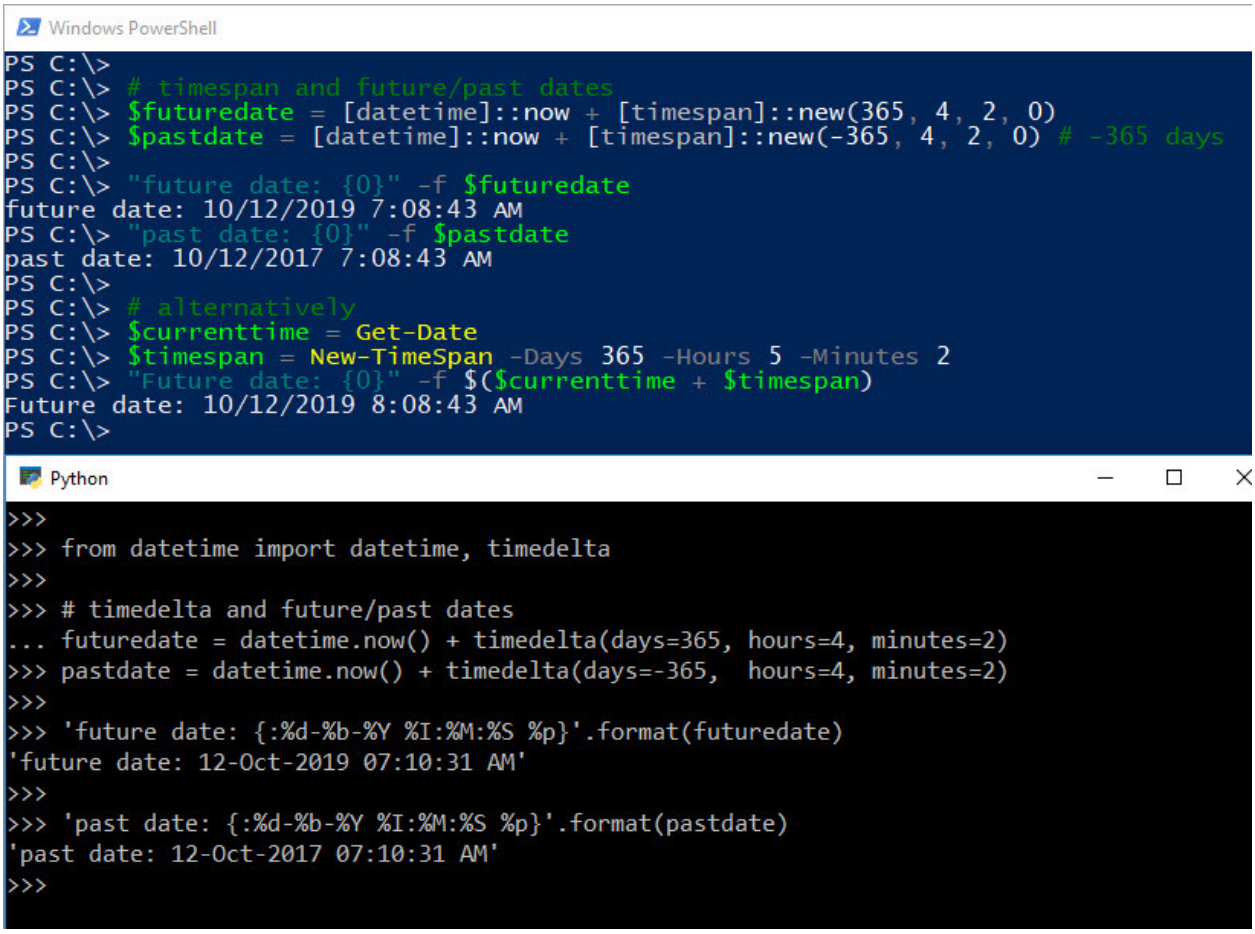
Time Span or Time Delta

In Powershell, a TimeSpan is a duration expressing the difference between two dates, time, or datetime and it could be used in multiple use cases like calculating a future or a past date.

```powershell
# timespan and future/past dates
$futuredate = [datetime]::now + [timespan]::new(365, 4, 2, 0)
$pastdate = [datetime]::now + [timespan]::new(-365, 4, 2, 0) # -365 days

"future date: {0}" -f $futuredate
"past date: {0}" -f $pastdate

# alternatively
$currenttime = Get-Date
$timespan = New-TimeSpan -Days 365 -Hours 5 -Minutes 2
"Future date: {0}" -f $($currenttime + $timespan)
```

Similarly, `TimeDelta()` is Python's implementation of a TimeSpan which works exactly like `[TimeSpan]` with slight syntax differences.

```python
from datetime import datetime, timedelta

# timedelta and future/past dates
futuredate = datetime.now() + timedelta(days=365, hours=4, minutes=2)
pastdate = datetime.now() + timedelta(days=-365,  hours=4, minutes=2)

'future date: {:%d-%b-%Y %I:%M:%S %p}'.format(futuredate)

'past date: {:%d-%b-%Y %I:%M:%S %p}'.format(pastdate)
```


Windows PowerShell
PS C:\>
PS C:\> # timespan and future/past dates
PS C:\> $futuredate = [datetime]::now + [timespan]::new(365, 4, 2, 0)
PS C:\> $pastdate = [datetime]::now + [timespan]::new(-365, 4, 2, 0) # -365 days
PS C:\>
PS C:\> "future date: {0}" -f $futuredate
future date: 10/12/2019 7:08:43 AM
PS C:\> "past date: {0}" -f $pastdate
past date: 10/12/2017 7:08:43 AM
PS C:\>
PS C:\> # alternatively
PS C:\> $currenttime = Get-Date
PS C:\> $timespan = New-TimeSpan -Days 365 -Hours 5 -Minutes 2
PS C:\> "Future date: {0}" -f $($currenttime + $timespan)
Future date: 10/12/2019 8:08:43 AM
PS C:\>
Python
>>>
>>> from datetime import datetime, timedelta
>>>
>>> # timedelta and future/past dates
... futuredate = datetime.now() + timedelta(days=365, hours=4, minutes=2)
>>> pastdate = datetime.now() + timedelta(days=-365, hours=4, minutes=2)
>>>
>>> 'future date: {:%d-%b-%Y %I:%M:%S %p}'.format(futuredate)
'future date: 12-Oct-2019 07:10:31 AM'
>>>
>>> 'past date: {:%d-%b-%Y %I:%M:%S %p}'.format(pastdate)
'past date: 12-Oct-2017 07:10:31 AM'
>>>

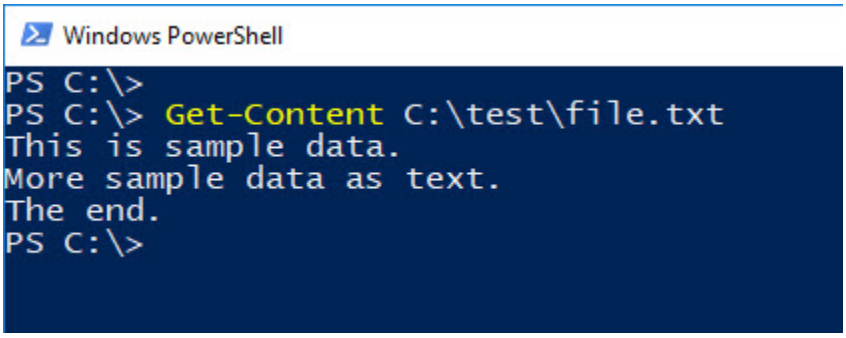
Chapter 11 - File Handling

In this chapter we are going to cover reading and writing data to files on you local machine using PowerShell and Python, by the end of this chapter you will be able to handle Read, Write and Append File operation. So let's get started!

Reading from a File

PowerShell offers a fairly simple approach to read files using the cmdlet: `Get-Content <file path>`, just provide the full or relative path to the cmdlet and the content of the file would be returned on your console.

```
Get-Content C:\test\file.txt
```



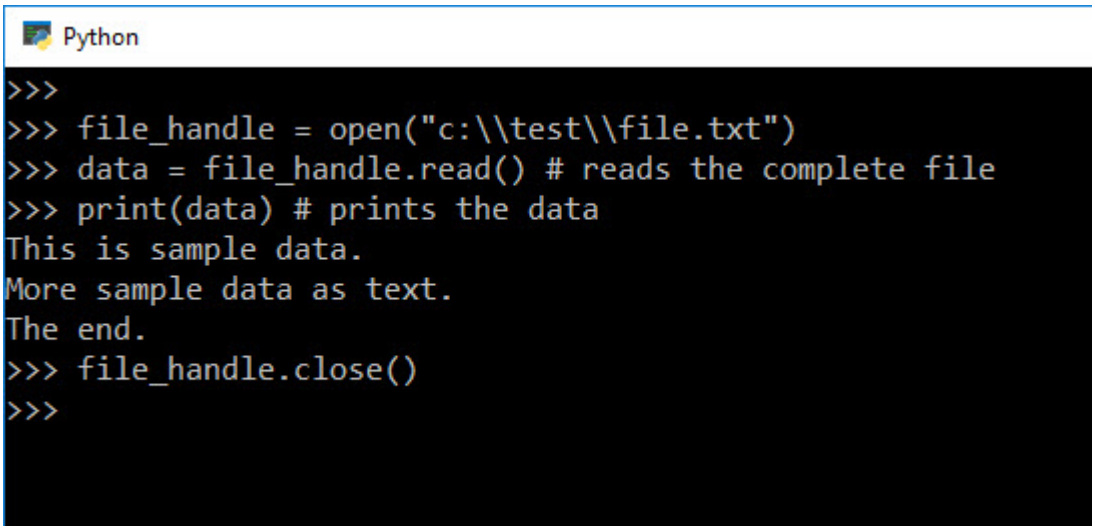
How to Read a File in owerShell

Whereas in Python, you've to use a built-in method `open()` which opens a file and returns stream of data in the file, but we have to make sure to escape any backslash character in the file path, otherwise the code will throw an `Invalid argument` error as (`\`) is also interpreted as an escape character in Python.

To know more about this Python built-in method: `help(open)` .

```
# escape the backslash in the file path
file_handle = open("C:\\test\\file.txt")
data = file_handle.read() # reads the complete file
print(data) # prints the data
file_handle.close()
```

In the above example, first we open the file in read-only mode which is the default mode of the `open()` method, now using the `read()` method capture the data stream from the file and store it in `data` variable. Later the `print()` method print this data on the Python console. It is important close the file handle so that another program can access the file, and also saves memory and avoid any program errors.

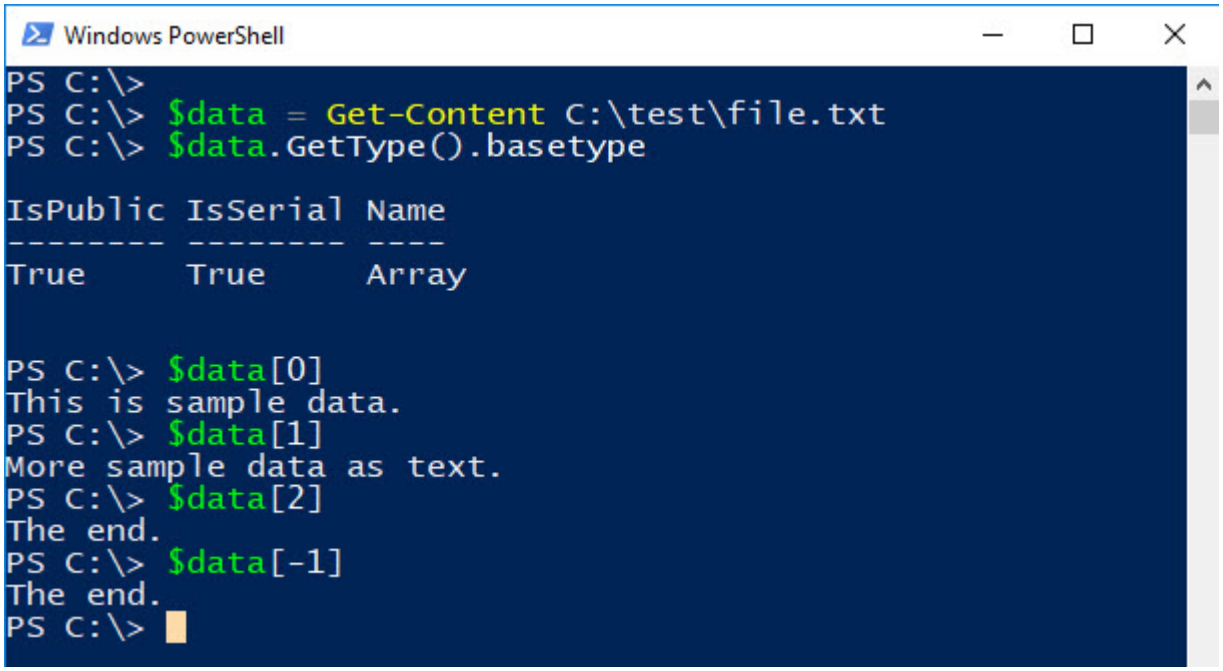


How to Read a File in Python

Reading the file line-by-line

When you capture data from a file using `Get-Content` cmdlet you get the complete content of the file, but in various scenarios it is required to read files in chunks or line-by-line. Data returned using this cmdlet in PowerShell is in form of an Array that means, the first element of the array is the first line of the file and you can use Array index values to access any line of the file.

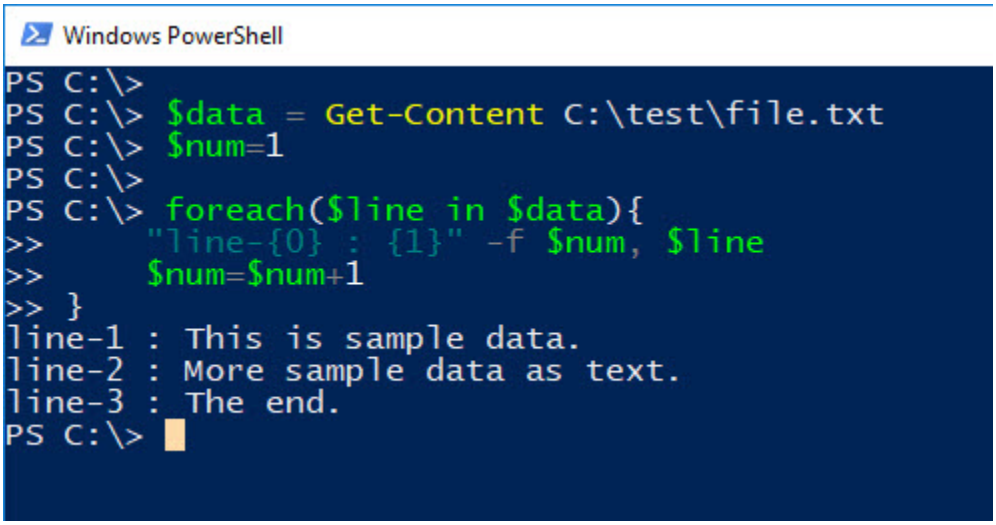
```
# capture content of the file
$data = Get-Content C:\test\file.txt

# get data type of the returned data
$data.GetType().basetype

# accessing file content line-by-line
$data[0] # first line
$data[1] # second line
$data[2] # third line
$data[-1] # last line
```



Reading files one line at a time in PowerShell

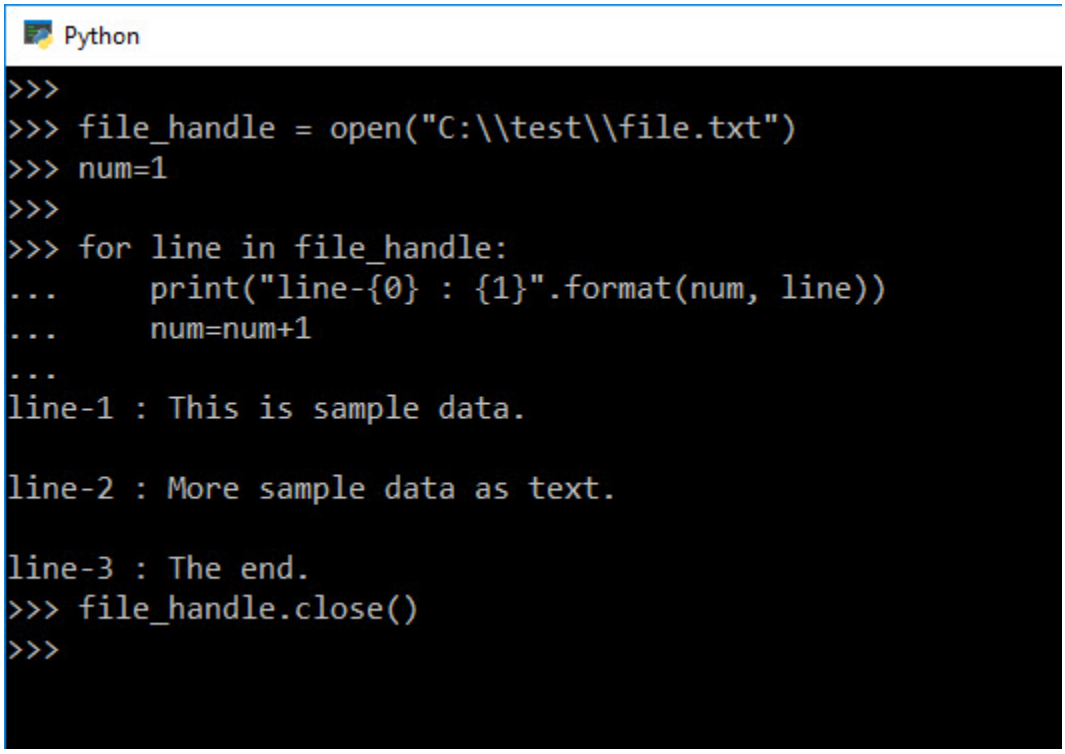
To read files one line at a time, you can also use a `ForEach` loop in PowerShell to iterate through each line of the file captured and stored as an array, like in the following example.

```
$data = Get-Content C:\test\file.txt
$num=1

foreach($line in $data){
    "line-{0} : {1}" -f $num, $line
    $num=$num+1
}
```



Likewise, a `for` loop could be used in Python to loop through each line of the text file.

```
file_handle = open("C:\\test\\file.txt")
num=1

for line in file_handle:
    print("line-{0} : {1}".format(num, line))
    num=num+1

file_handle.close()
```



File Modes

In one of the previous examples, before reading a file we opened it using Python's built-in `open()` method, that returns a file object which would be utilized to call other supporting methods associated with file. This method has the following Syntax:

```
file object = open(file_name [, access_mode][, buffering])
```

We've already looked into the file name parameter, and buffering parameter is to enable buffering action with the indicated buffer size, but here I want to discuss about something important that is the Access mode parameters.

The optional parameter `access_mode` determines the mode in which the file has to be opened, that is, read, write or append, following is a complete list of possible mode values. By default it is set to 'r' that is the `read-only` mode, due to this it was not required to define this parameter explicitly in previous examples while reading the files.

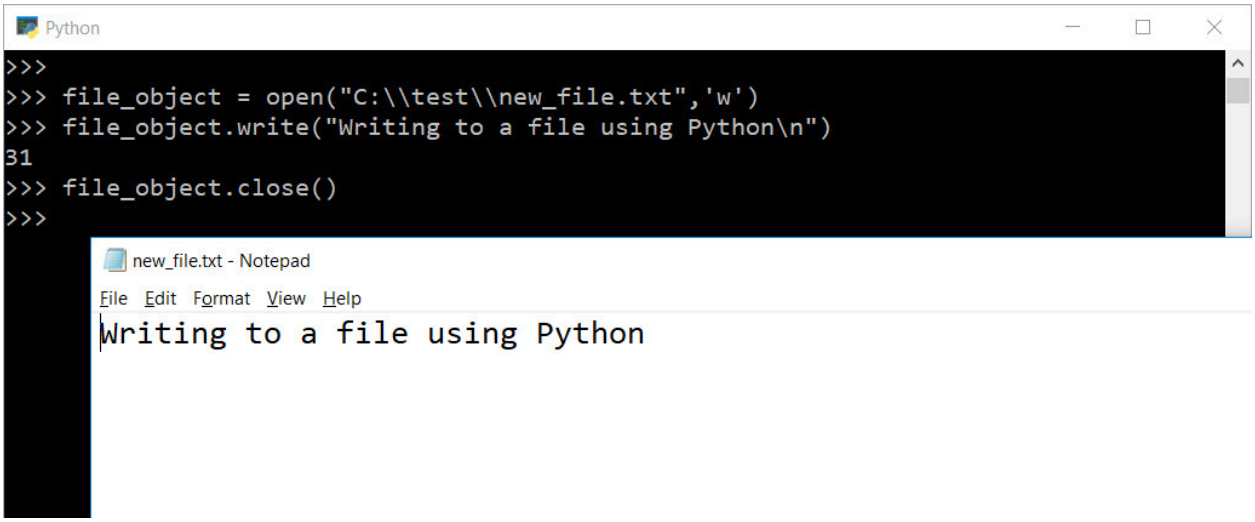
Operation	Mode	Description
Read	r	Opens a file for reading only and is the default mode.
Read	rb	Opens a file for reading only in binary format.
Read	r+	Opens a file for both reading and writing
Read	rb+	Opens a file for both reading and writing in binary forma.
Write	w	Opens a file for writing only.
Write	wb	Opens a file for writing only in binary format.
Write	w+	Opens a file for both writing and reading.
Write	wb+	Opens a file for both writing and reading in binary format.
Append	a	Opens a file for appending. If the file does not exist, it creates a new file for writing.
Append	ab	Opens a file for appending in binary format. If the file does not exist, it creates a new file for writing.
Append	a+	Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
Append	ab+	Opens a file for both appending and reading in binary format. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

Create and write to a text File

To create a file in Python we basically open a file in Write mode: '`w`', if in case the file doesn't exists it will create a new file, otherwise it would be overwritten without any warnings

```
1 file_object = open("C:\\test\\new_file.txt",'w')
2 file_object.write("Writing to a file using Python\n")
3 file_object.close()
```

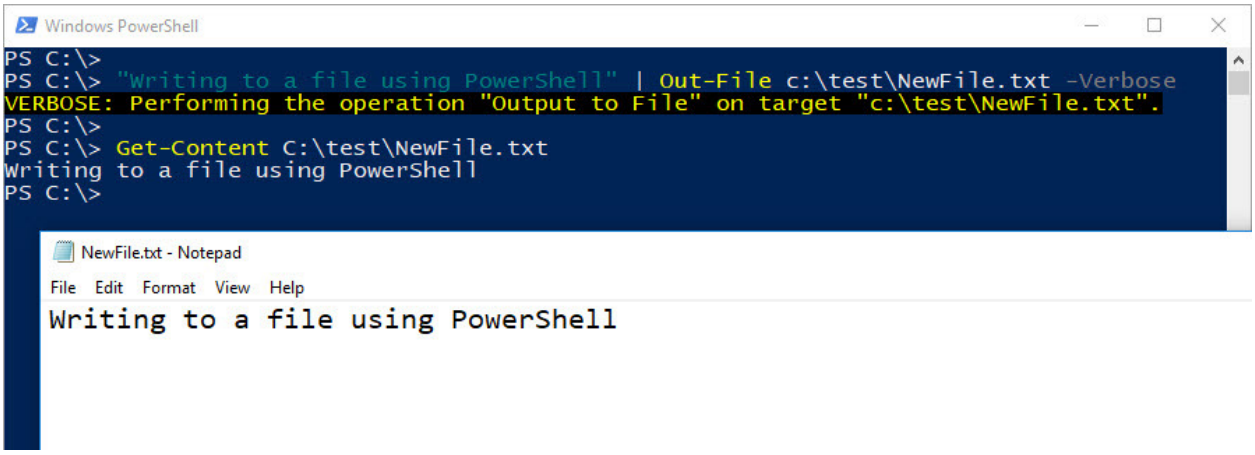
In the above example, when we used `open()` function and provided a file path that does not exist. Then a new file is created in `write` mode, which is later written using the `write()` function, and returns the number of characters (`31`) written, in other words it is the length of the string written to the file. Finally we close the file once the write operation is complete.



Writing to a file in Python

Compared to Python, in PowerShell writing to a file is very simple, and we can simply pipe a string to `Out-File` cmdlet and provide a file path. If the file does not exist, then a new file is created otherwise it will be overwritten.

```
1  "Writing to a file using PowerShell" | Out-File NewFile.txt
```



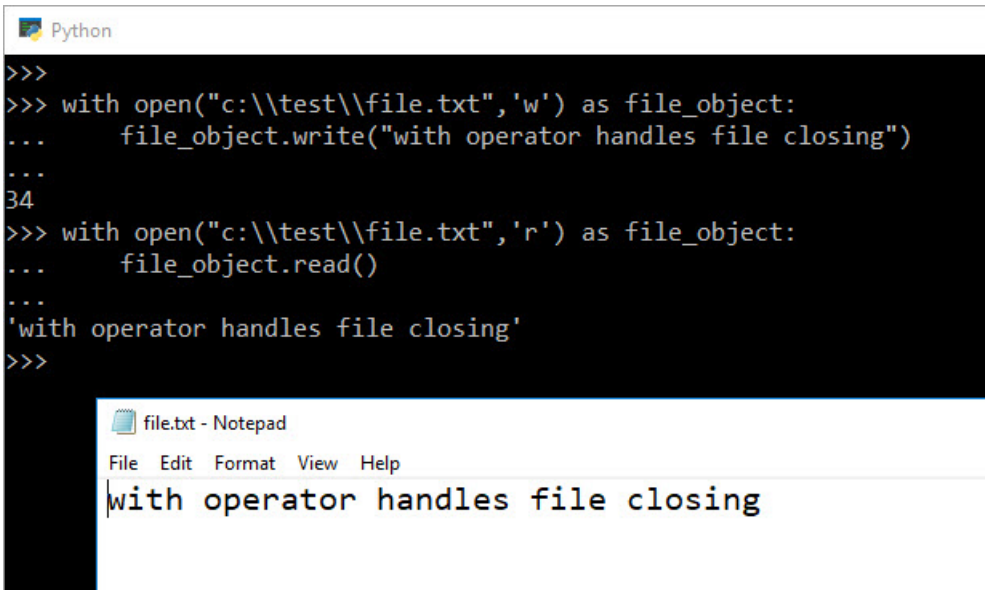
Writing to a file in PowerShell

In the above PowerShell example, unlike Python we don't have to close the file once the write operation is complete which is very useful, but Python overcomes this drawback using the '`with`' operator which can simplify the read-write operations in Python.

The '`with`' operator creates a context manager that automatically handles closing of file, let's see how this works:

```
# writing to a file
with open("c:\\test\\file.txt",'w') as file_object:
    file_object.write("with operator handles file closing")

# reading from a file
with open("c:\\test\\file.txt",'r') as file_object:
    file_object.read()
```

In the above example you don't have to explicitly define the `close()` function.



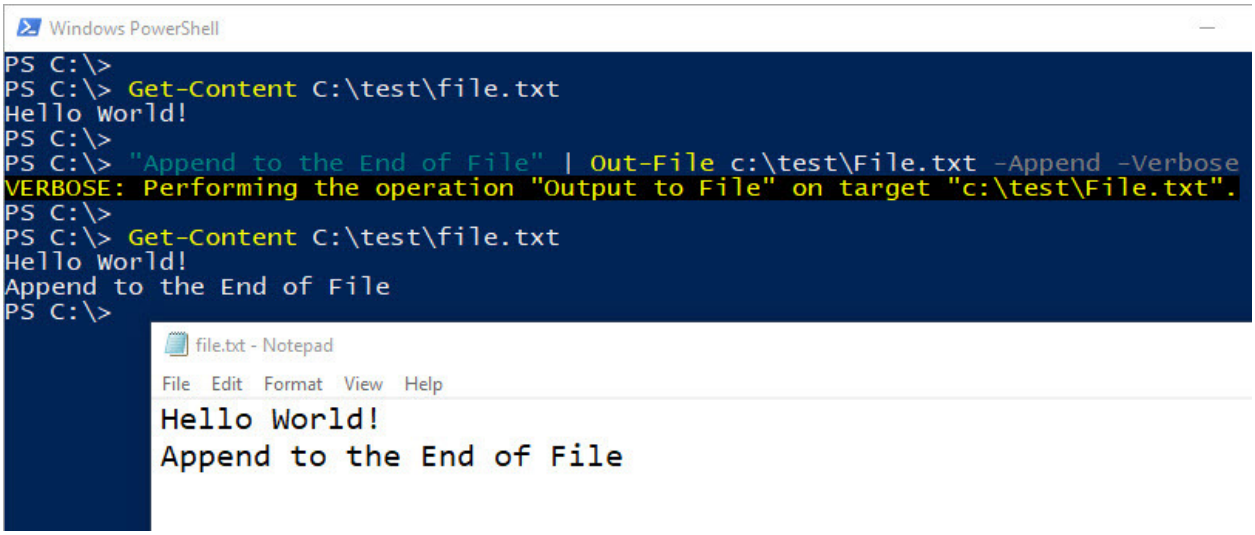
Using the 'with' operator

Appending data to the File

The append operation adds a string to the end of the written document.

In PowerShell we use the `Out-File` cmdlet with the '`-Append`' switch parameter to append a string at the end of the file.

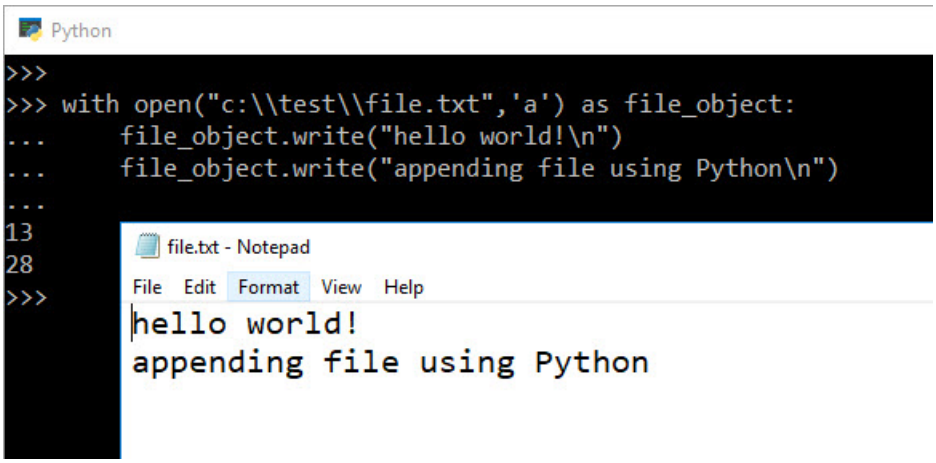
```
"Append this string to the End of File" | Out-File File.txt -Append
```



Appending a file in PowerShell

On other hand, to append a file using Python, you've to first open the file in `append` mode: 'a' and then write the content to the file using `write()` function, which positions the file pointer now at the end of the file.

```
1  with open("c:\\test\\file.txt",'a') as file_object:
2      file_object.write("hello world!\n")
3      file_object.write("appending file using Python\n")
```



Appending a file in Python

Chapter 12 - Arrays, List, ArrayList and Tuples

I this chapter we will begin with PowerShell and Python data structures, in fact we are going to discuss about the most popular data structures like: Arrays and Lists

Arrays

A PowerShell Array is immutable (fixed sized) Data Structure that can hold heterogeneous elements.

Arrays in Powershell are implicitly typed, but in case you want to create a strongly typed array, cast the variable as an array type, such as `string[]`, `long[]`, or `int32[]`

```
# creating arrays
$array = 1,2,3,4,5 # homogeneous array
$array = 1, 2, 'text', 1.5 # heterogeneous array
$array.gettype() # get data type of the array

[int[]] $array = 1,2,3,4,5 # String typed [int[]] array

# Strongly typed array can cast different data types of elements to the defined data\
 type
# if the conversion is possible like [double] to [int] or [char] to [int]
[int[]] $array = 1,2.1,3,4.3,5,[char]'a'

# throws error 'Cannot convert value "text" to type "System.Int32".'
# because [string] can't be type casted to [int]
[int[]] $array = 1, 2, 'text', 1.5
```

Whereas, Python doesn't have a native array Data Structure, but Arrays are supported using the built-in module called '`array`', which is required to be imported before using this data structure. The elements stored in a Python array are restricted or constrained by their data type, hence are Homogeneous in nature. That means a character array can only store character elements, unlike PowerShell Arrays that can hold heterogeneous elements.

To define an Array in Python you have to specify the data type using a 'type code' during the array creation, which is a single character like in the following table:

Type code	C Type	Python Type	Minimum size in bytes
'c'	char	character	1
'b'	signed char	int	1
'B'	unsigned char	int	1
'u'	Py_UNICODE	Unicode character	2
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	long	2
'l'	signed long	int	4
'L'	unsigned long	long	4
'f'	float	float	4
'd'	double	float	8

```
# import the module, since Array is not a native data structure
import array
a = array.array("I",[1,2,3,4,5]) # homogeneous, strong typed array
type(a) # get data type of the array
# not implicitly type casted, you've to explicitly typecast elements with the array \
data type
a = array.array("I",[1,2,3,int(4.3),5])
# throws error 'TypeError: integer argument expected, got float'
a = array.array("I",[1,2,3,4.3,5])

# operations on array
a.insert(1,7) # inserting elements
a.pop(3) # delete and return an element
a.reverse() # reverse the array
```

Key Pointers

- PowerShell Arrays
  * Immutable
  * Homogeneous or Heterogeneous
  * Loosely typed
  * Built-in
- Python Arrays
  * Mutable
  * Homogeneous
  * Strictly typed
  * Not built-in and have to be imported
  * Generally comparatively faster and efficient than arrays (* though that depends on use cases)

ArrayList (.Net)

In the first section of this chapter, we discussed PowerShell's default arrays which are very convenient and useful because they are meant to be flexible and allows you to store heterogeneous objects of any data type including `$null` in a single array.

```
$array = 'hello world!', 111, 13.14, (Get-Date), $null
```

But this flexibility also introduces some drawbacks, that are:

* You can not ensure that all elements are of a specific type or should be converted to a single data type, because of the heterogenous nature of the elements. Since PowerShell has access to the `.NET` type system, you can overcome these drawbacks, by creating an array that is restricted to the specific type of interest, using a cast or type-constrained variable:

```
    [int[]] $Array = 1, 2, 3
```

- PowerShell Arrays are immutable in nature that means they are of fixed size and once defined elements can not be added. To overcome this you should add the new element to the existing elements of the array, and assign all elements (new + existing) to an Array variable.

```
  # array
  $fruits = @()

  # adding elements to arrays
  # throws an exception "Collection was of a fixed size."
  # because powershell array variables are immutable and this is an empty array
  $fruits.Add('apple')
  $fruits.IsFixedSize # output:True
  # so when we are assigning an element the size will change,
  # which is not allowed

  # adding elements the correct way, this works!
  # because in the background it creates a whole new array
  # that includes the new value and then discards the old array.
  $fruits += 'apple'
  $fruits += 'banana'
  $fruits += 'orange'
```

PowerShell can also use .Net Class: `System.Collections.ArrayList`, which are mutable, dynamic and much faster in nature.

```powershell
# arralist
$names = New-Object System.Collections.ArrayList
[void] $names.Add('Prateek')
[void] $names.Add('Singh')
[void] $names.Add('Prateek')
[void] $names.Add('Singh')
$names.IsFixedSize # output:False
```

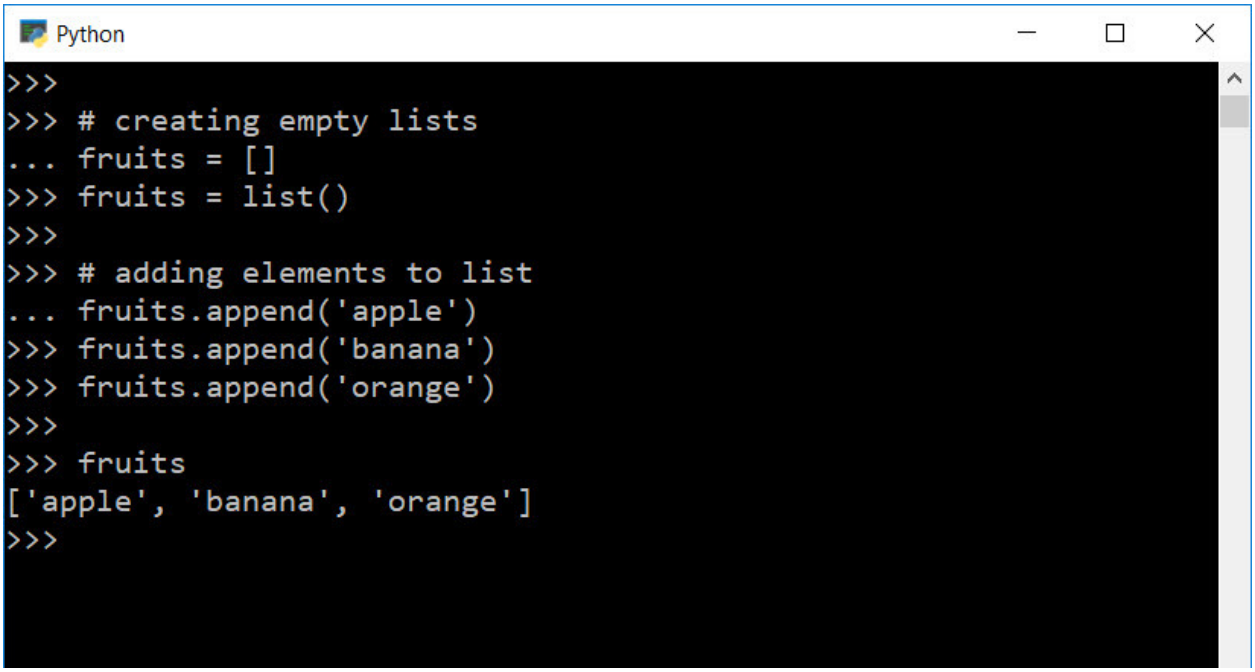
Key Pointers

- PowerShell (.NET) ArrayList
- Mutable
- Homogeneous or Heterogeneous
- Loosely typed
- Built-in

Lists

A list in Python is just an ordered collection of heterogeneous item types and is dynamically mutable, i.e size can change. Lists data structures support operations like: `add`, `delete`, `insert`.

```python
# creating empty lists
fruits = []
fruits = list()

# adding elements to list
fruits.append('apple')
fruits.append('banana')
fruits.append('orange')
```



Python Lists

Key Pointers

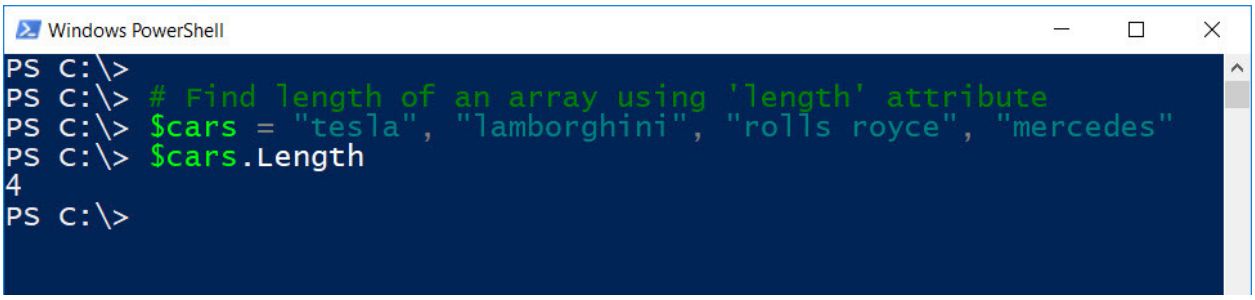
- Python List
- Mutable
- Heterogeneous or homogenous items hence can be used as Arrays
- Loosely typed
- Built-in into Python
- Generally comparatively slower than arrays, though the performance may vary in different use cases

Common PowerShell Array and Python Lists Operations

Finding Length

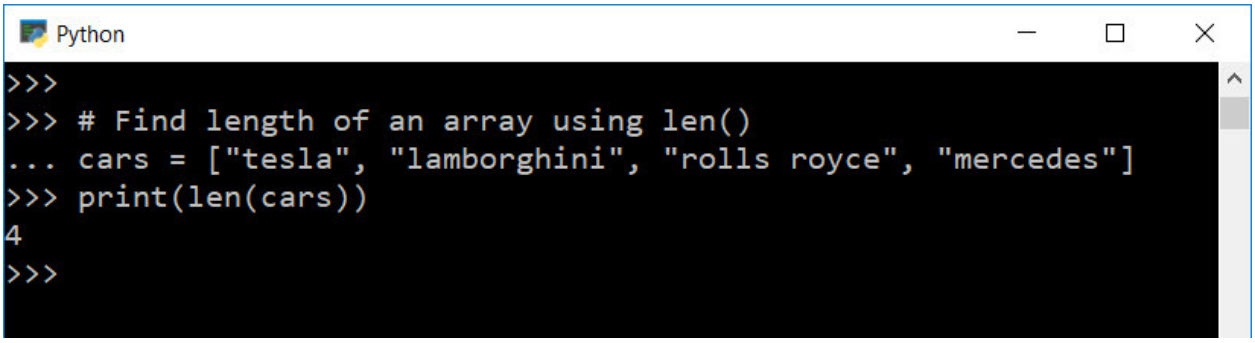
PowerShell Arrays have a property 'Length' which can be accessed using the ('.') Dot operator to get the Length (number of elements) of the array

```powershell
# Find length of an array using 'length' attribute
$cars = "tesla", "lamborghini", "rolls royce", "mercedes"
$cars.Length
```



Whereas, Python Array's length is calculated using the Built-in function called `len()`

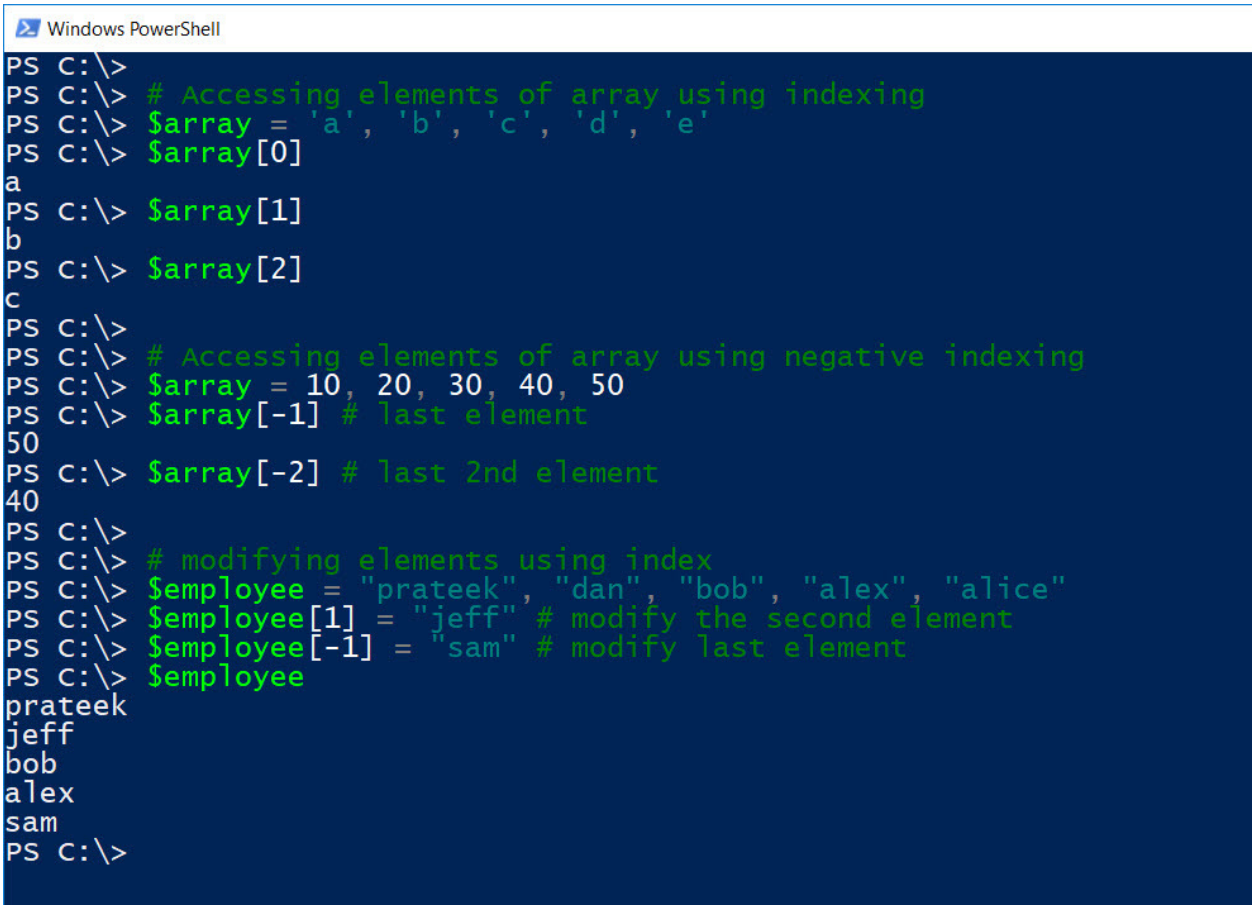
```python
# Find length of an array using len()
cars = ["tesla", "lamborghini", "rolls royce", "mercedes"]
print(len(cars))
```



Indexing and access/modify elements

PowerShell and Python are 'Zero indexed', that means the first element of a PowerShell Array or Python List has index '`0`'. Array/List index can be used in square brackets `[index]` to access and modify elements like in the following examples. Modifying an element is technically Variable assignment to an array/list index.

```
# Accessing elements of array using indexing
$array = 'a', 'b', 'c', 'd', 'e'
$array[0]
$array[1]
$array[2]

# Accessing elements of array using negative indexing
$array = 10, 20, 30, 40, 50
$array[-1] # last element
$array[-2] # last 2nd element

# modifying elements using index
$employee = "prateek", "dan", "bob", "alex", "alice"
$employee[1] = "jeff" # modify the second element
$employee[-1] = "sam" # modify last element
$employee
```



Array Indexing in PowerShell

```python
# Index is the position of element in an list.
# In Python, lists are zero-indexed.
# This means, the element's position starts with 0 instead of 1.
# Accessing elements of list using indexing
array = ['a', 'b', 'c', 'd', 'e']
print(array[0])
print(array[1])
print(array[2])

# Accessing elements of list using negative indexing
array = [10, 20, 30, 40, 50]
print(array[-1])  # last element
print(array[-2])  # last 2nd element

# modifying elements using index
employee = ["prateek", "dan", "bob", "alex", "alice"]
employee[1] = "jeff"  # modify the second element
employee[-1] = "sam"  # modify last element
print(employee)
```

Python

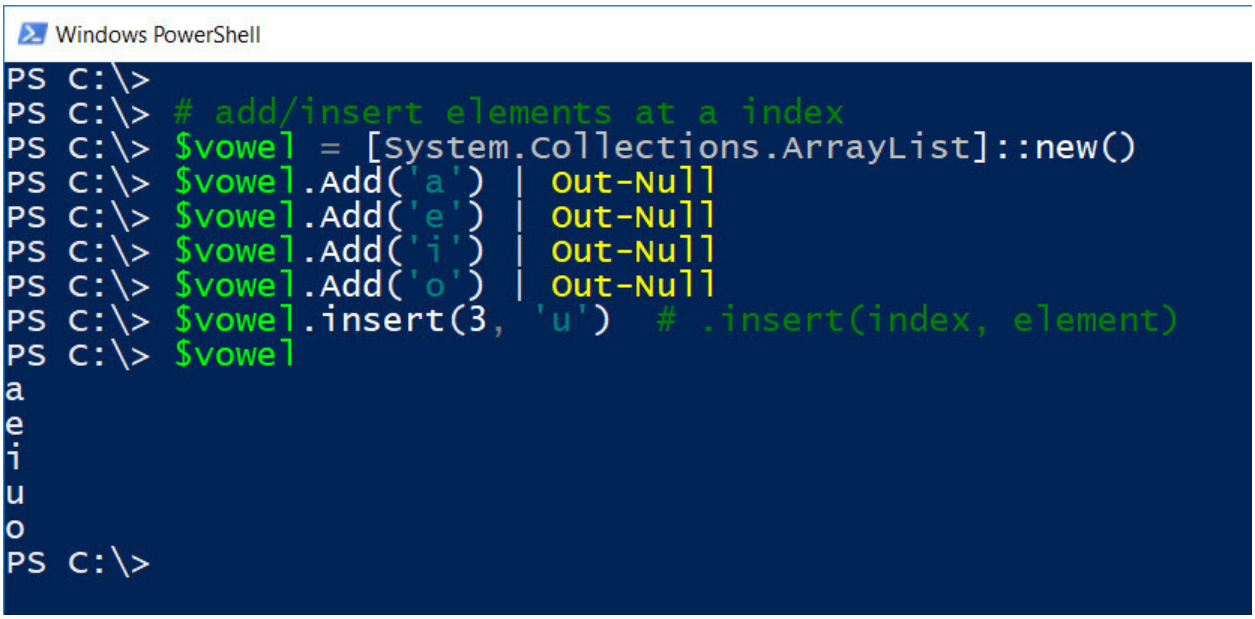
```
>>>
>>> # Index is the position of element in an list.
... # In Python, lists are zero-indexed.
... # This means, the element's position starts with 0 instead of 1.
... # Accessing elements of list using indexing
... array = ['a', 'b', 'c', 'd', 'e']
>>> print(array[0])
a
>>> print(array[1])
b
>>> print(array[2])
c
>>>
>>> # Accessing elements of list using negative indexing
... array = [10, 20, 30, 40, 50]
>>> print(array[-1])  # last element
50
>>> print(array[-2])  # last 2nd element
40
>>>
>>> # modifying elements using index
... employee = ["prateek", "dan", "bob", "alex", "alice"]
>>> employee[1] = "jeff"  # modify the second element
>>> employee[-1] = "sam"  # modify last element
>>> print(employee)
['prateek', 'jeff', 'bob', 'alex', 'sam']
>>>
```

Array Indexing in Python

Insert/Add elements

Adding an element in PowerShell ArrayList is done by `add()` method, whereas element insertion is achieved by the `insert()` method in both PowerShell and Python.

```powershell
# add/insert elements at a index
$vowel = [System.Collections.ArrayList]::new()
$vowel.Add('a') | Out-Null
$vowel.Add('e') | Out-Null
$vowel.Add('i') | Out-Null
$vowel.Add('o') | Out-Null
$vowel.insert(3, 'u')  # .insert(index, element)
$vowel
```



Adding/Inserting element in PowerShell

Python List uses `append()` method to add elements at the end (last index) of the List.

```python
# append/insert elements at a index
vowel = []  # or just vowel = ['a','e','i','o']
vowel.append('a')
vowel.append('e')
vowel.append('i')
vowel.append('o')
vowel.insert(3, 'u')  # .insert(index, element)
# insertion doesn't replace the value on the index,
# but shift the following indices
print(vowel)
```

Python

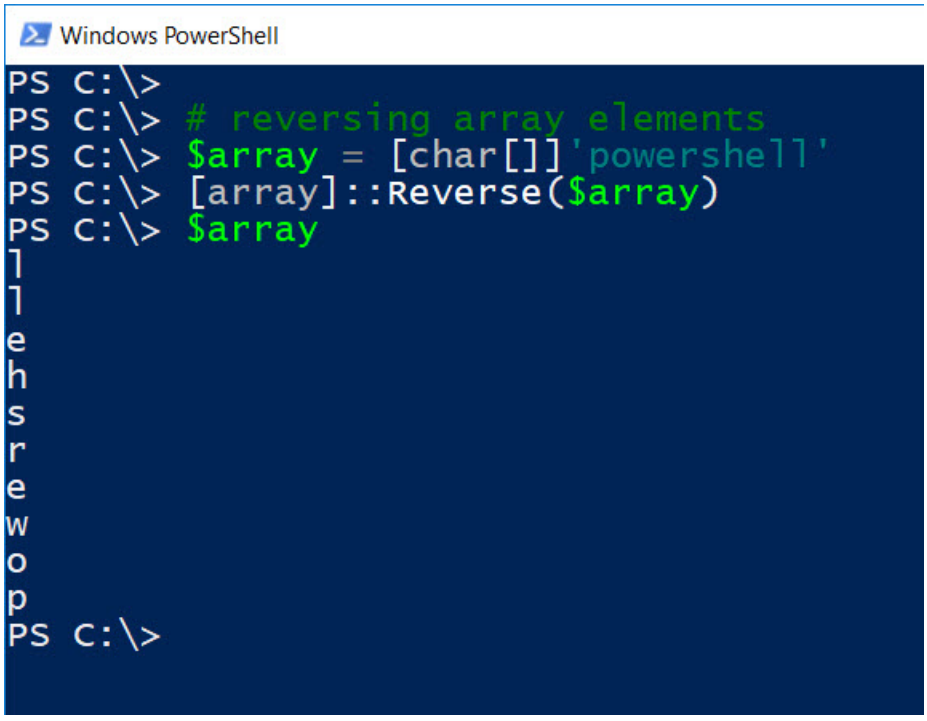
```
>>>
>>> # append/insert elements at a index
... vowel = []  # or just vowel = ['a','e','i','o']
>>> vowel.append('a')
>>> vowel.append('e')
>>> vowel.append('i')
>>> vowel.append('o')
>>> vowel.insert(3, 'u')  # .insert(index, element)
>>> # insertion doesn't replace the value on the index,
... # but shift the following indices
... print(vowel)
['a', 'e', 'i', 'u', 'o']
>>>
```

Adding/Inserting element in Python

Reverse elements

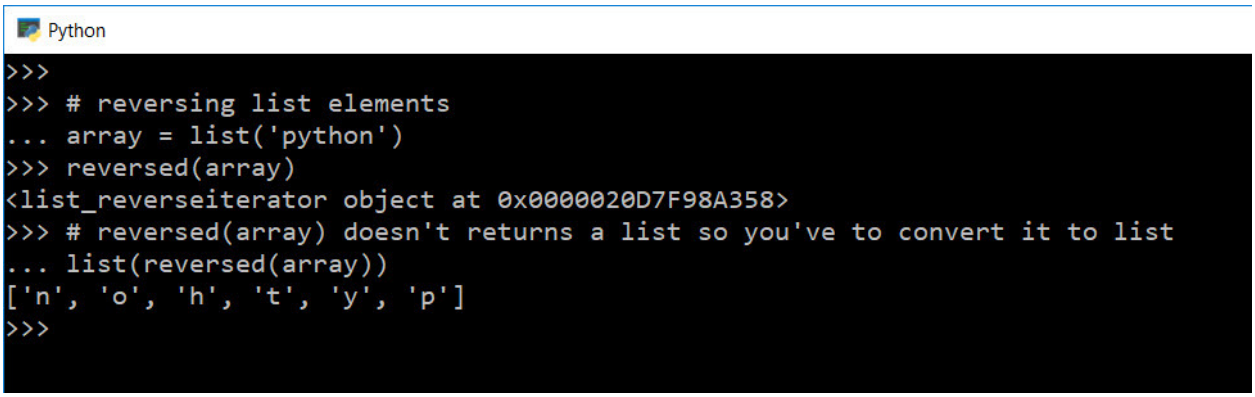
PowerShell `[Array]` Type accelerator has a built-in method to `reverse()` the elements of an array.

```
# reversing array elements
$array = [char[]]'powershell'
[array]::Reverse($array)
$array
```



Similarly, Python has a `reversed()` method which can reverse the order of List elements.

```
# reversing list elements
array = list('python')
reversed(array)
# reversed(array) doesn't returns a list so you've to convert it to list
list(reversed(array))
```

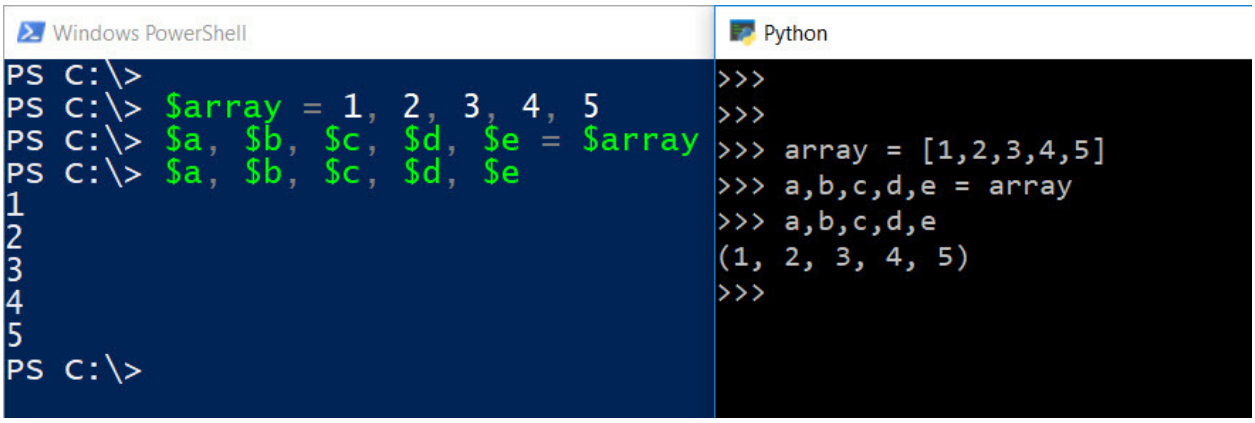


One to Many: Array to the Variable assignment

PowerShell and Python both offers One to Many: Array/List to variable assignment, like in the following code samples:

```powershell
# assigning array elements to multiple variables
$array = 1, 2, 3, 4, 5
$a, $b, $c, $d, $e = $array
```

```python
# assigning list elements to multiple variables
array = [1,2,3,4,5]
a,b,c,d,e = array
```

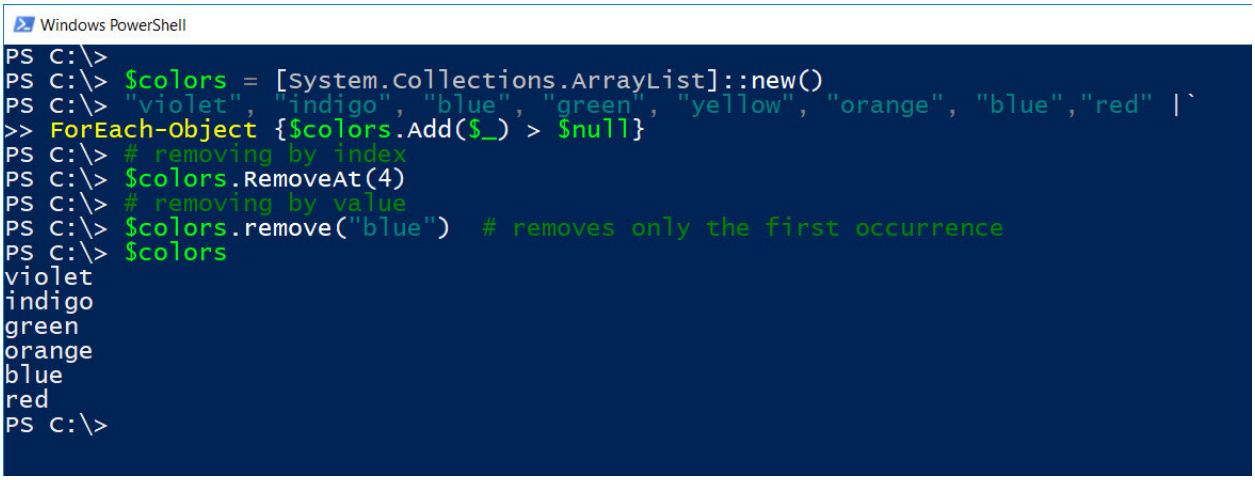


Remove elements

PowerShell and Python both allows you to remove Array/List elements at a specific index, and by value, following are some examples

```powershell
$colors = [System.Collections.ArrayList]::new()
"violet", "indigo", "blue", "green", "yellow", "orange", "blue","red" |`
ForEach-Object {$colors.Add($_) > $null}
# removing by index
$colors.RemoveAt(4)
# removing by value
$colors.Remove("blue")  # removes only the first occurrence
$colors
```

In the above example, PowerShell employs `RemoveAt()` method available in `ArrayList` class to remove an item a specific index, but to remove items by value, the `Remove()` method is used, that removes the first occurrence of the target element value.




```
# delete elements of an array using del, remove() and pop()
colors = ["violet", "indigo", "blue", "green", "yellow", "orange", "red"]
# deleting by index
del colors[4]
# deleting by value
colors.remove("blue") # removes only the first occurrence
colors.pop(3) # .pop(index) deletes and returns the element
print(colors)
```

Whereas, in Python a delete statement (‘`del`’) is used to remove\delete a target list or list index, which follows the syntax:

```
del target_list[index]
```

But to delete a list item by a specific value, the `remove()` method is used. If you want to return the deleted element you can use `pop()` method instead as demonstrated in the following example:

Python

```
>>>
>>> # delete elements of an array using del, remove() and pop()
... colors = ["violet", "indigo", "blue", "green", "yellow", "orange", "red"]
>>> # deleting by index
... del colors[4]
>>> # deleting by value
... colors.remove("blue") # removes only the first occurrence
>>> colors.pop(3) # .pop(index) deletes and returns the element
'orange'
>>> print(colors)
['violet', 'indigo', 'green', 'red']
>>>
```

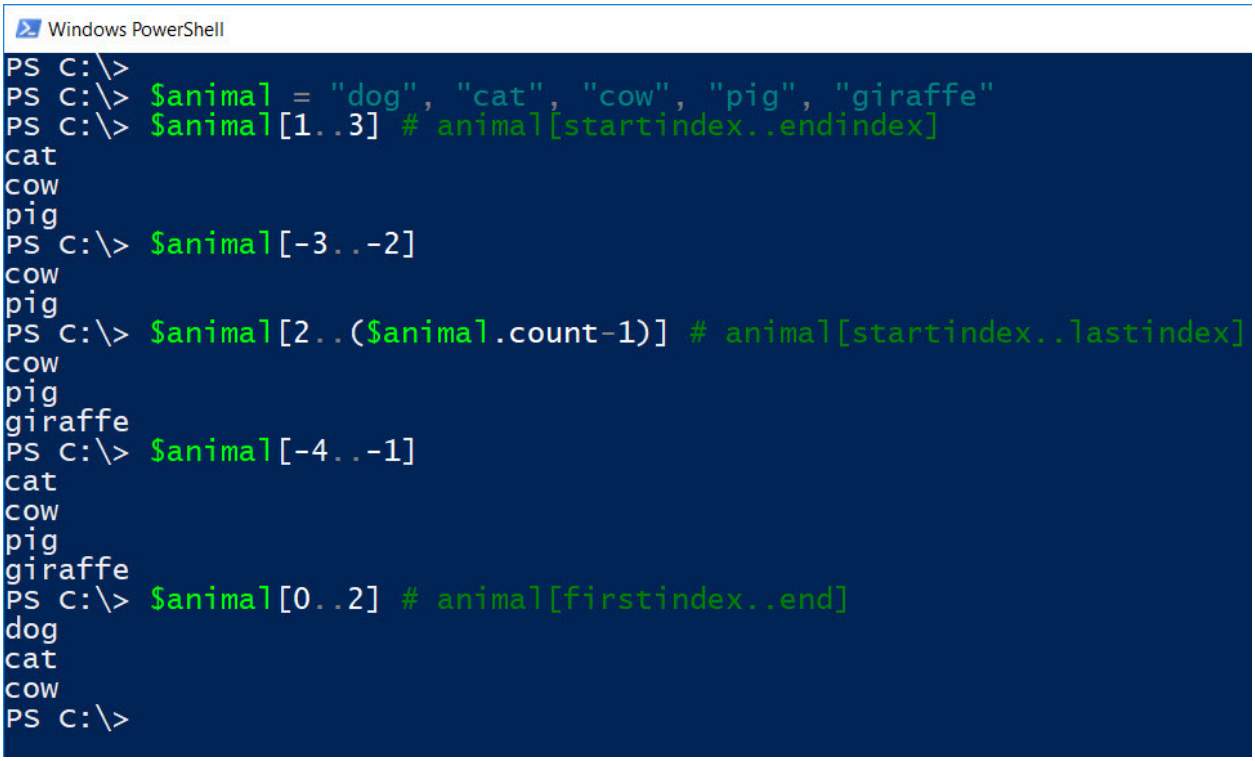
Array/List slicing

Array slicing in PowerShell and Python is just like slicing a String which we covered in one of our previous chapters.

PowerShell syntax-

```
array[startindex..endindex]
```

```
# array slicing
$animal = "dog", "cat", "cow", "pig", "giraffe"
$animal[1..3] # animal[startindex..endindex]
$animal[-3..-2]
$animal[2..($animal.count-1)] # animal[startindex..lastindex]
$animal[-4..-1]
$animal[0..2] # animal[firstindex..end]
```



Python syntax-

```
array[startindex:end:step]
```

Here,
end = EndIndex+1
step = Increments

```
# slicing lists
animal = ["dog", "cat", "cow", "pig", "giraffe"]
print(animal[1:4]) # animal[start:end] items start through end-1
print(animal[-3:-1])
print(animal[2:]) # animal[start:] items start through the rest of the array
print(animal[-4:])
print(animal[:3]) # animal[:end] items from the beginning through end-1
print(animal[:]) # copy of the whole array
print(animal[1:4:2]) # animal[start:end:step]  start through not past end, by step
```

Python

```
>>>
>>> # slicing lists
... animal = ["dog", "cat", "cow", "pig", "giraffe"]
>>> print(animal[1:4]) # animal[start:end] items start through end-1
['cat', 'cow', 'pig']
>>> print(animal[-3:-1])
['cow', 'pig']
>>> print(animal[2:]) # animal[start:] items start through the rest of the array
['cow', 'pig', 'giraffe']
>>> print(animal[-4:])
['cat', 'cow', 'pig', 'giraffe']
>>> print(animal[:3]) # animal[:end] items from the beginning through end-1
['dog', 'cat', 'cow']
>>> print(animal[:]) # copy of the whole array
['dog', 'cat', 'cow', 'pig', 'giraffe']
>>> print(animal[1:4:2]) # animal[start:end:step]  start through not past end, by step
['cat', 'pig']
>>>
```

Multi Dimensional Array/List

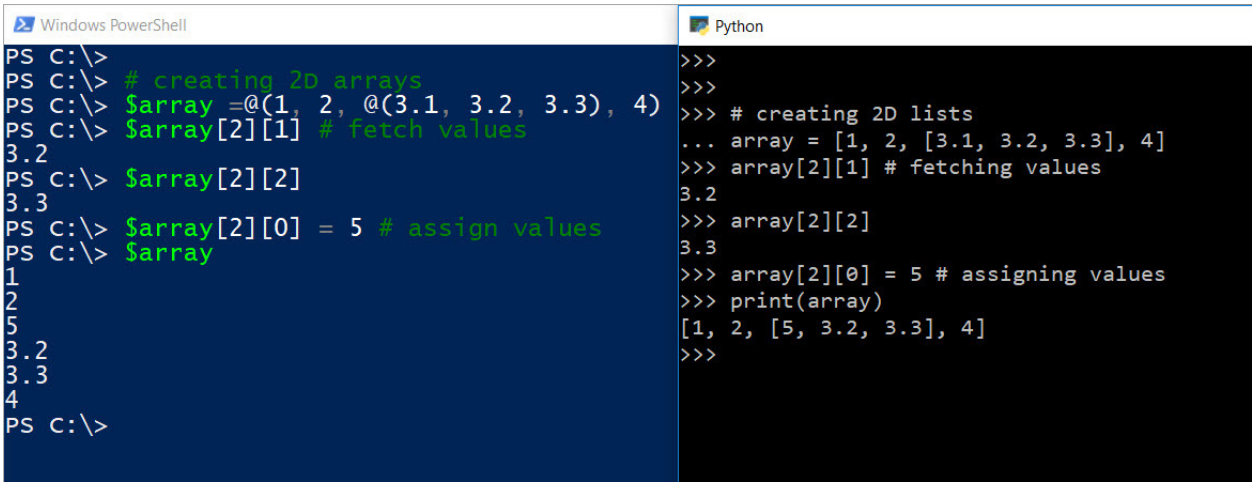
Python and PowerShell both enables you to create multi-dimensional arrays and syntax to create one is almost same, like in the following examples:

Two Dimensional Arrays

```
# creating 2D arrays
$array =@(1, 2, @(3.1, 3.2, 3.3), 4)
$array[2][1] # fetch values
$array[2][2]
$array[2][0] = 5 # assign values
$array
```

Two Dimensional Lists

```
# creating 2D lists
array = [1, 2, [3.1, 3.2, 3.3], 4]
array[2][1] # fetching values
array[2][2]
array[2][0] = 5 # assigning values
print(array)
```



Tuples

Tuples are immutable (constant) Data Structure support in Python and PowerShell, unlike lists or array lists which can be modified.

Once a Tuple is defined you cannot delete, add or edit any values inside it. This can be very helpful in use cases where you might pass the control to someone but you do not want them to edit or manipulate data in your data structure, but rather maybe just perform operations separately in a copy of this Data Structure.

PowerShell:

Defining Tuples in PowerShell requires you to use .NET classes or [System.Tuple] type accelerator

```
# tuples
$tuple = [System.Tuple]::Create(5,6,7,8)
$tuple
$tuple.Item1
```

Python:

Whereas, use `()` parenthesis to define a tuple in Python.

```
tuple = (5, 6, 7, 8) # () parenthesis to create tuples
print('Tuple:', tuple)
# number in square bracket is index of element you want to access
print("Tuple index 1:", tuple[1])
```

Windows PowerShell

```
PS C:\>
PS C:\> # tuples
PS C:\> $tuple = [System.Tuple]::Create(5,6,7,8)
PS C:\> $tuple


Item1  : 5
Item2  : 6
Item3  : 7
Item4  : 8
Length : 4



PS C:\> $tuple.Item1
5
PS C:\>
```

Python

```
>>>
>>> tuple = (5, 6, 7, 8) # () parenthesis to create tuples
>>> print('Tuple:', tuple)
Tuple: (5, 6, 7, 8)
>>> # number in square bracket is index of element you want to access
... print("Tuple index 1:", tuple[1])
Tuple index 1: 6
>>>
```

Sets

Sets are a collection of distinct (unique) objects that only hold unique values in the data set. A set is an unordered collection but a mutable one (can be modified) and are very helpful when going through a huge data set and want to find Intersection, union and other data sets.

PowerShell utilizes .Net class `System.Collections.Generic.HashSet<T>` to create a HashSet

```powershell
# define a set
$x = [System.Collections.Generic.HashSet[string]]::new()
'python'.ToCharArray().foreach({[void]$x.add($_)})

$y = [System.Collections.Generic.HashSet[string]]::new()
'powershell'.ToCharArray().foreach({[void]$y.add($_)})

$x.ExceptWith($y) # All the elements in x but not in y
# union
$x.UnionWith($y) # Unique elements in x or y or both
# intersection
$x.IntersectWith($y) # Elements in both x and y
$x.SymmetricExceptWith($y) # Elements in x or y but not in both
```

Whereas, Python has an inbuilt method `set()` to define sets.

```python
# define sets
x = set('python')
y = set('powershell')

print(x - y) # All the elements in x but not in y
# union
print(x | y) # Unique elements in x or y or both
# intersection
print(x & y) # Elements in both x and y
print(x ^ y) # Elements in x or y but not in both
```

Chapter 13 - Dictionary and Hashtable

In this chapter, you'll learn about PowerShell Hashtables and Python dictionary; how they are created, accessed, adding and removing elements and some built-in methods.

Hash Table

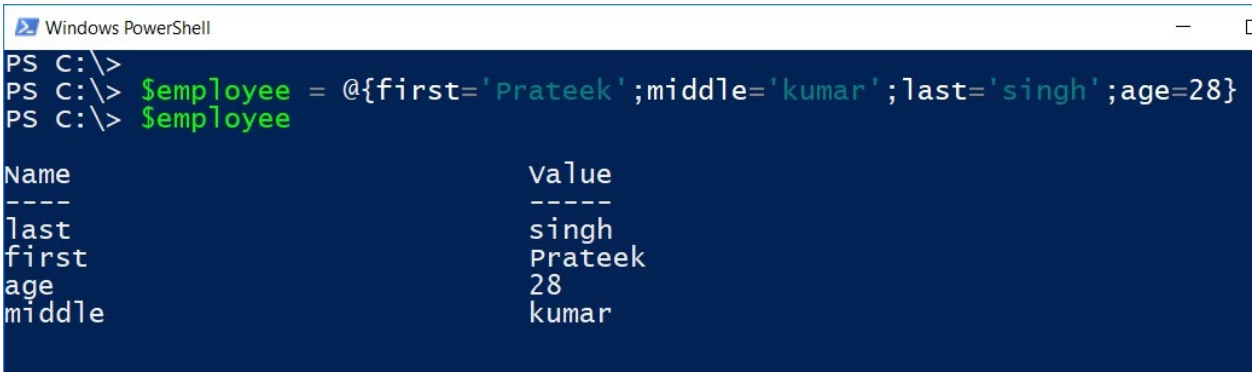
A hash table in PowerShell is also known as a dictionary or an associative array, is a data structure that stores one or more key-value pairs. To create a hash table, we use an at `@` symbol followed by open and close braces. Inside the braces you can define a key-value pair separated by '=' operator, in case there are multiple key-value pairs you separate them by a semi-colon ';'.

Syntax:

```
@{key1=value1; key2=value2; key3=value3 ....}
```

For example, a hash table might contain details like, first, middle, last name and age of an employee.

```
$employee = @{first='Prateek';middle='kumar';last='singh';age=28}
```



A Hash Table in PowerShell

Dictionary

A dictionary in Python is a unordered, changeable collection which stores indexed key-value pairs. In Python dictionaries are written with curly brackets. Creating a dictionary is as simple as placing items inside curly braces `{ }` separated by comma.

A key-value pair is separated by a colon ' : '

Syntax:

```
employee = {key1:value1, key2:value2, key3:value3 ....}
```

While values can be of any data type and even can repeat but each key in a dictionary must be unique. For example lets see how employee details can be stored in a Python Dictionary:

```
employee = {'first':'Prateek', 'middle':'kumar', 'last':'singh', 'age':28}
```

```
Python
>>>
>>> employee = {'first':'Prateek', 'middle':'kumar', 'last':'singh', 'age':28}
>>> employee
{'first': 'Prateek', 'middle': 'kumar', 'last': 'singh', 'age': 28}
>>>
```

A Dictionary in Python

Accessing Key-Value pair

A key value pair of hashtable\dictionary in PowerShell and Python is accessed by the name followed by key inside square `[]` brackets.

PowerShell:

```
# accessing elements of hashtable
$employee['first']
```

Python:

```
# accessing elements of dictionary
print(employee['first'])
```

Nested HashTable and Dictionary

Hashtables and Dictionaries are data structures that map a key to a value, but often it is smart to store a hashtable as a value inside another hashtable where a key pointing to it, this is called nested hash tables or dictionaries. Nested dictionaries can come handy to store and retrieve information easily and fast.

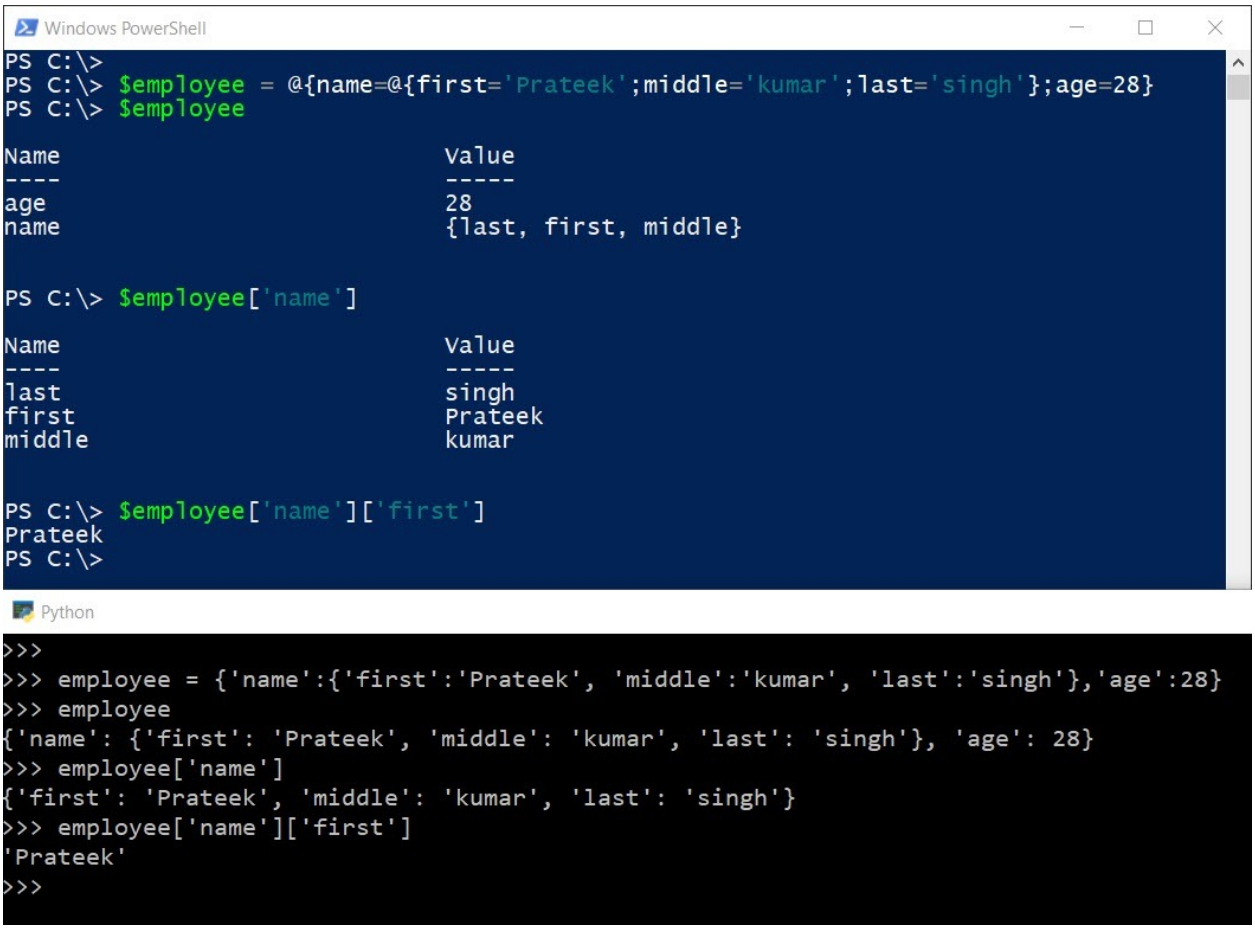
Lets see how we can store employee details as a nested dictionary, to do so instead of storing first, middle and last names separately, we store them in a hashtable and them map it to a key `name` in another hashtable, so that it is accessible from the parent hashtable like in the following code sample.

PowerShell:

```
$employee = @{name=@{first='Prateek';middle='kumar';last='singh'};age=28}
```

Python:

```
employee = {'name':{'first':'Prateek', 'middle':'kumar', 'last':'singh'},'age':28}
```



Nested HashTable and Dictionary

Adding Key-Value pair

In Powershell and Python you can map values to keys using variable assignment, like in the below code sample a new key-value pair would be creates if it doesn't exist, otherwise the value would be overwritten.

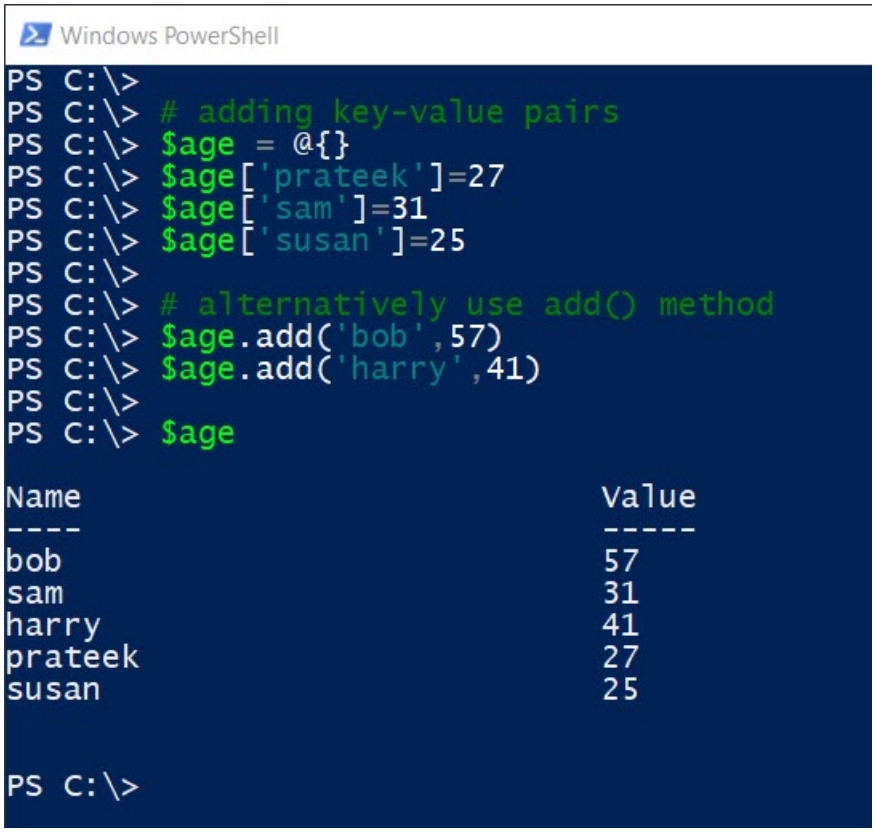
Syntax:

```
$name[key] = value
```

PowerShell hashtable is a `System.Collections.Hashtable` object and you can use the properties and methods of Hashtable objects in PowerShell. Like `add()` method can also be utilized to add key-value pairs to a Hash Table.

PowerShell:

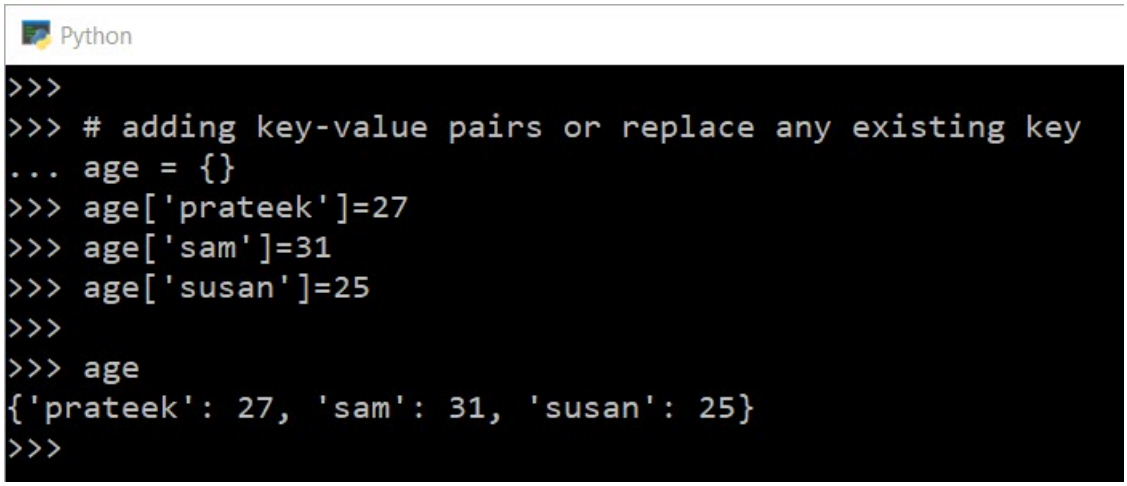
```
# adding key-value pairs
$age = @{}
$age['prateek']=27
$age['sam']=31
$age['susan']=25

# alternatively use add() method
$age.add('bob',57)
$age.add('harry',41)
```



Adding key-value pair in a PowerShell Hashtable

Python:

```
# adding key-value pairs or replace any existing key
age = {}
age['prateek']=27
age['sam']=31
age['susan']=25
```



Adding key-value pair in a PowerShell Hashtable

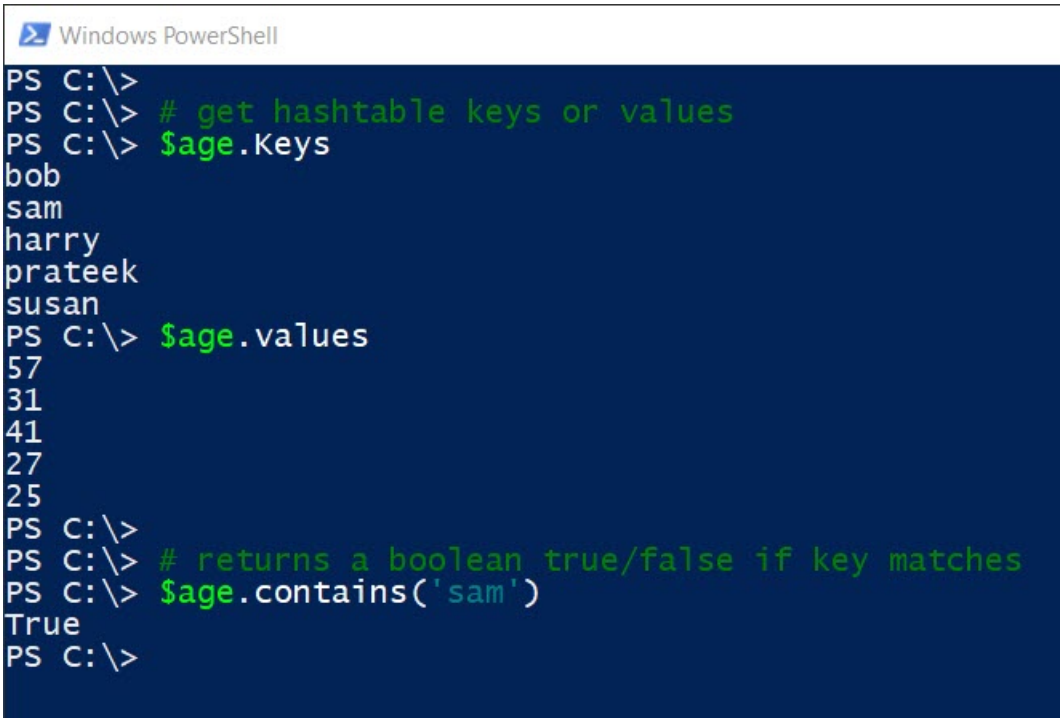
Finding Keys and Values

Hashtables and Dictionaries in PowerShell and Python are objects, to find keys or values use '`key`' and '`value`' property in PowerShell, and `key()` and `value()` methods in Python.

To search a specific key in a dictionary, `contains('key')` and `__contains__('key')` method are used in PowerShell and Python Respectively, which will return boolean `True` if the key exists.

PowerShell:

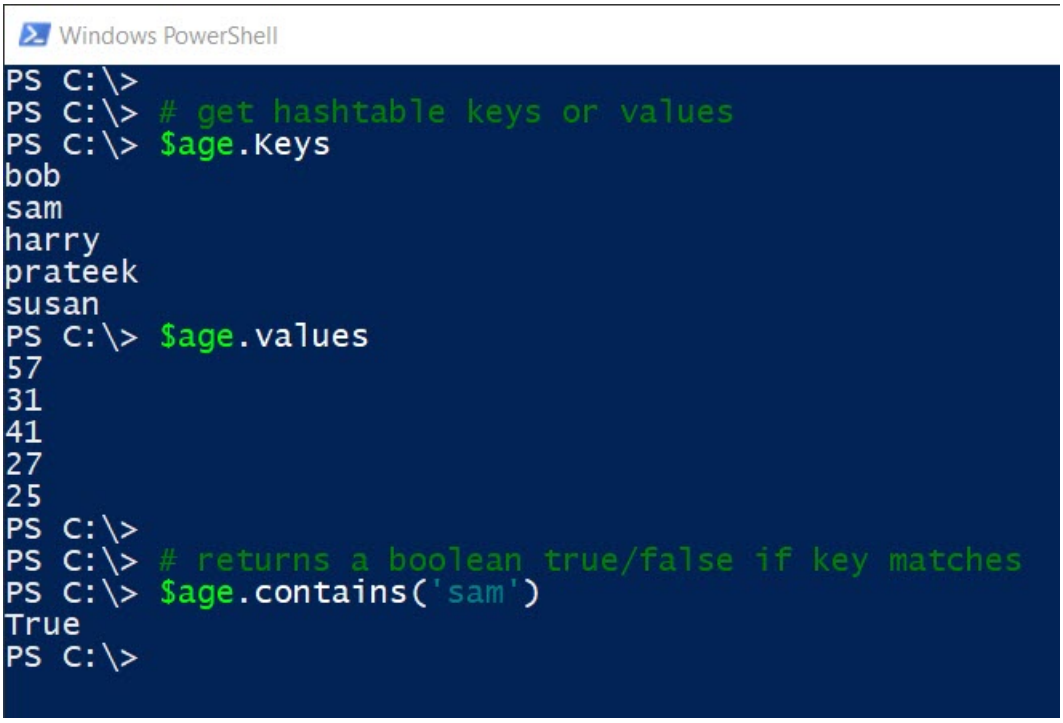
```
# get hashtable keys or values
$age.Keys
$age.values

# returns a boolean true/false if key matches
$age.contains('sam')
```



Finding key, value in PowerShell Hashtable

Python:

```
# get dictionary keys or values
age.keys()
age.values()

# returns a boolean true/false if key matches
age.__contains__('sam')
```



Finding key, value in PowerShell Hashtable

Iterating through key-value pairs

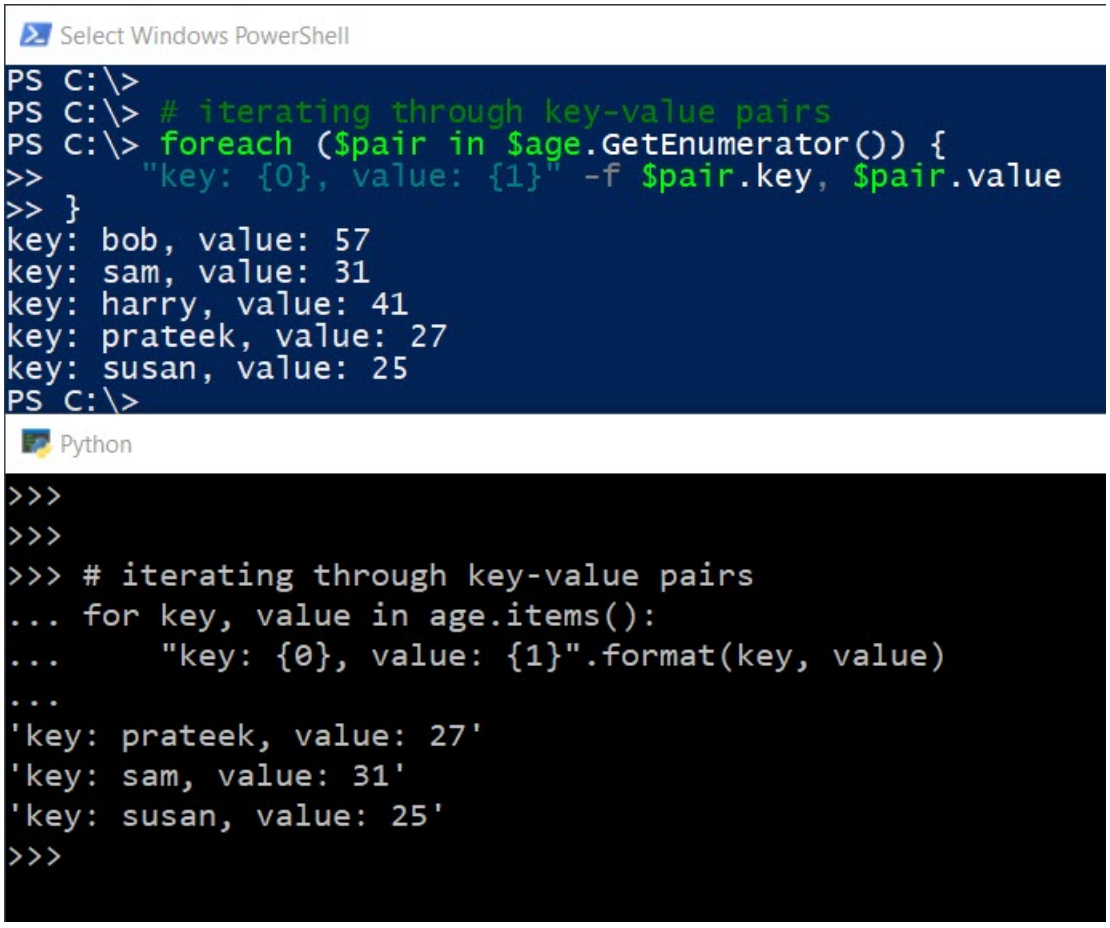
In some use cases you'll want to iterate key-value pairs in a hashtable or a dictionary, to demonstrate that following are some examples

In PowerShell we utilize the `GetEnumerator()` method to iterate through each key-value pair of the Hashtable.

```
# iterating through key-value pairs
foreach ($pair in $age.GetEnumerator()) {
    "key: {0}, value: {1}" -f $pair.key, $pair.value
}
```

Whereas, in Python same can be done using the `items()` method.

```
# iterating through key-value pairs
for key, value in age.items():
    "key: {0}, value: {1}".format(key, value)
```

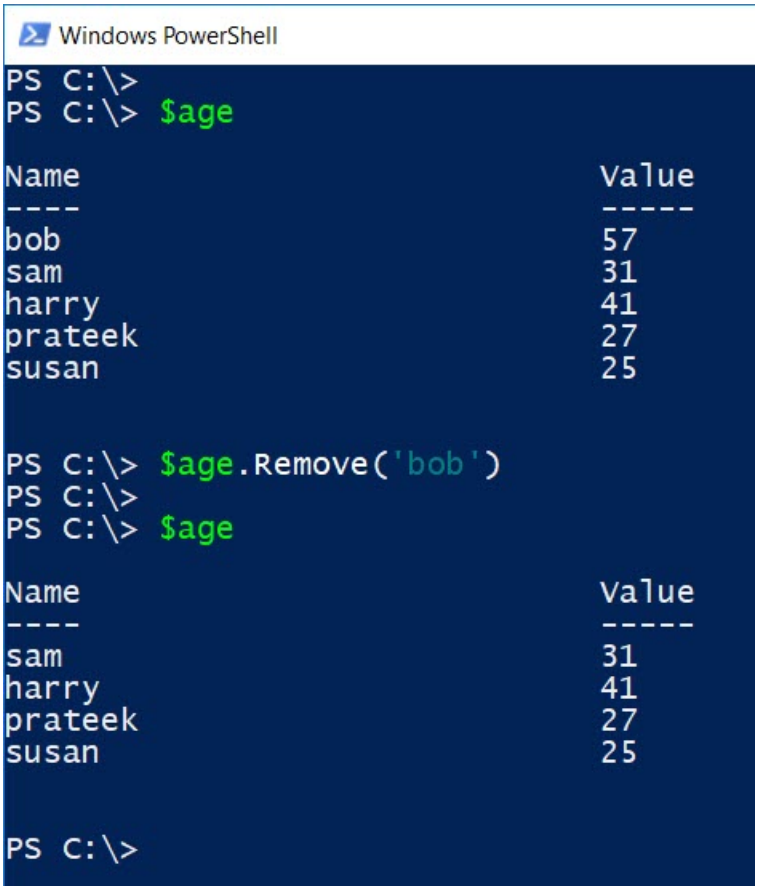


Iterating through key-value pairs of a dictionary

Deleting a key-value pair

Deleting a key-value pair in PowerShell Hashtable is as easy as using the `remove('key')` method with name of the target key, like in the following example:

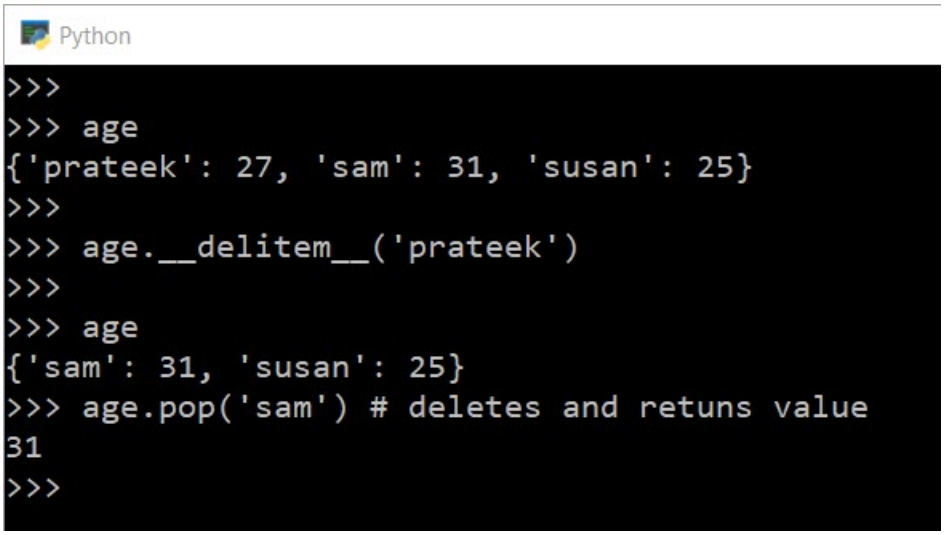
```
# create a hashtable
$age = @{}
$age['prateek']=27
$age['sam']=31
$age['susan']=25

# delete a key-value pair
$age.remove('prateek')
```



Delete a key-value pair in PowerShell

Whereas, Python has various methods to delete a key-value pair, like `__delitem__('key')` method is used to delete a target key-value pair, and `pop('key')` method can delete the the key-value pair but it will also return the `value` of deleted key-value pair.

```
# create a hashtable
age = {}
age['prateek']=27
age['sam']=31
age['susan']=25

# delete a key-value pair
age.__delitem__('prateek')

# deletes and returns value
age.pop('sam')
```

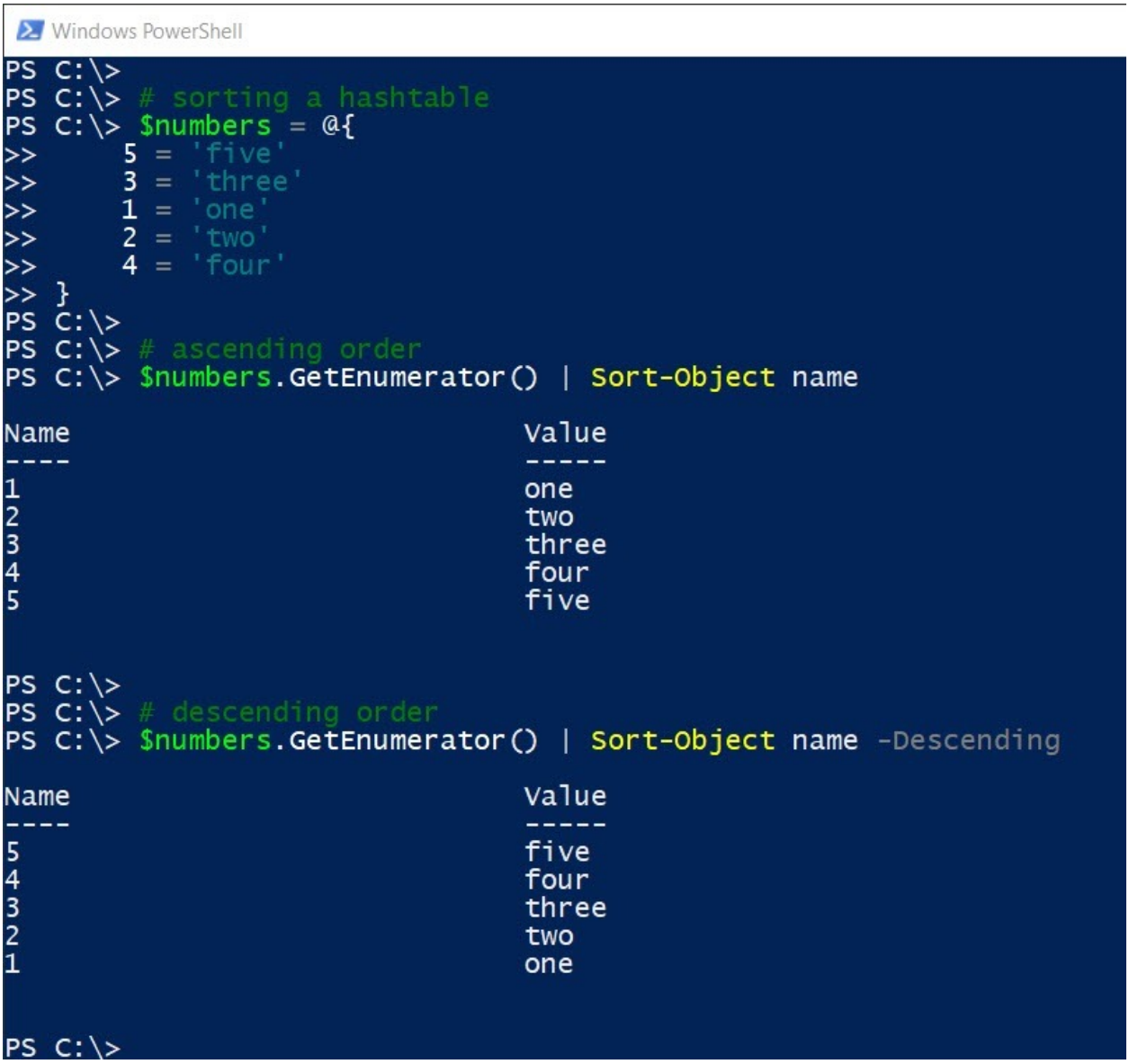


Delete a key-value pair in Python

Sorting hashtable

Some scenarios requires sorted Hashtables and Dictionaries; In PowerShell `GetEnumerator()` method is used to get individual key-value pairs of the HashTable and the if we pipe it to `Sort-Object` cmdlet, to sort it by property 'name' then key-value pairs would be returned which are sorted by the value of the hashtable keys.

```
# sorting a hashtable
$numbers = @{
    5 = 'five'
    3 = 'three'
    1 = 'one'
    2 = 'two'
    4 = 'four'
}

# ascending order
$numbers.GetEnumerator() | Sort-Object name

# descending order
$numbers.GetEnumerator() | Sort-Object name -Descending
```



Sorting HashTables

Whereas, in Python we use `items()` method to get all key-value pairs and the using the built-in method `sorted()` we sort the items of the Dictionary.

```python
# sorting a dictionary
numbers = {
    5 : 'five',
    3 : 'three',
    1 : 'one',
    2 : 'two',
    4 : 'four'
}

# ascending order
dict(sorted(numbers.items()))

# descending order
dict(sorted(numbers.items(),reverse=True))
```

```
Python
>>>
>>> # sorting a dictionary
... numbers = {
...     5 : 'five',
...     3 : 'three',
...     1 : 'one',
...     2 : 'two',
...     4 : 'four'
... }
>>>
>>> # ascending order
... dict(sorted(numbers.items()))
{1: 'one', 2: 'two', 3: 'three', 4: 'four', 5: 'five'}
>>>
>>> # descending order
... dict(sorted(numbers.items(),reverse=True))
{5: 'five', 4: 'four', 3: 'three', 2: 'two', 1: 'one'}
>>>
```

Sorting Dictionaries

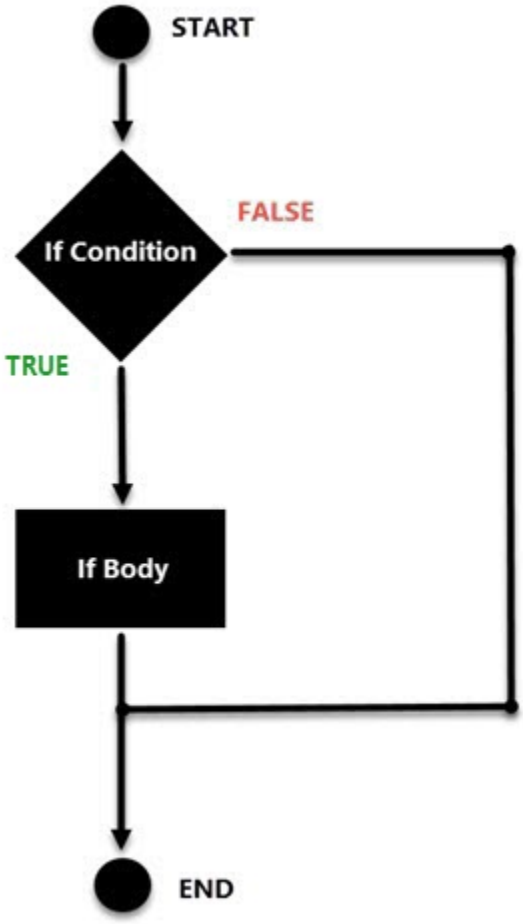
Chapter 14 - Conditional Statements

In programming, it is very often required to check some conditions and change the behavior of the program. So in this chapter, we are going to learn about syntax and concepts Conditional statements in Python and PowerShell.

If Statement

Conditional statements are part of every programming language, which runs or does not runs a part of code depending upon the condition defined in the program, by evaluating conditions. Programs that run based on whether or not those conditions are met are called conditional code.

Let's begin with the `if` statement which is the first building block of a conditional statement, used to evaluate whether a statement was `True` or `False` and only executes the body of the `if` statement if the condition was `True`, following diagram represents the flow control of an if statement:



Syntax of `if` statement in Python:

```
if expression:
   statement
```

Let's take an example in Python where we have a variable that has an integer value assigned to it, and we want a portion of program to run if and only if the integer value is between a specific range, or positive/negative or Zero.

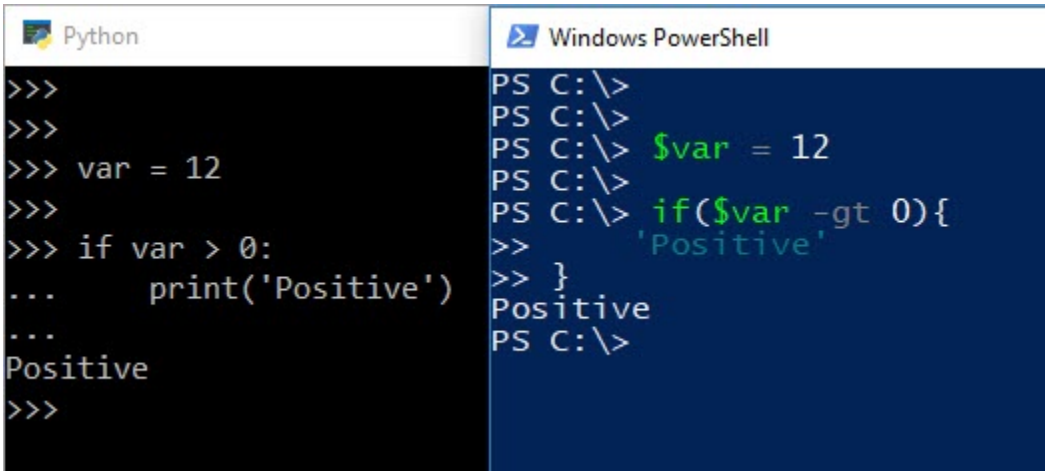
```
var = 12

if var > 0:
    print('Positive')

if var < 0:
    print('Negative')

if var == 0:
    print('Zero')
```

A similar implementation in PowerShell is following with little bit of syntax changes in the `if` statement and how differently PowerShell employs the logical operators: `-gt` (greater than), `-lt` (less than), `-eq` (equals to). Python as a programming language minds spaces and indentation in the sytax, but in PowerShell all that matters is that you enclose the condition in a parenthesis `( )` and the body of `if` statement inside a set of braces `{ }`

Syntax in PowerShell:

```
if(expression){
    statement
}
```

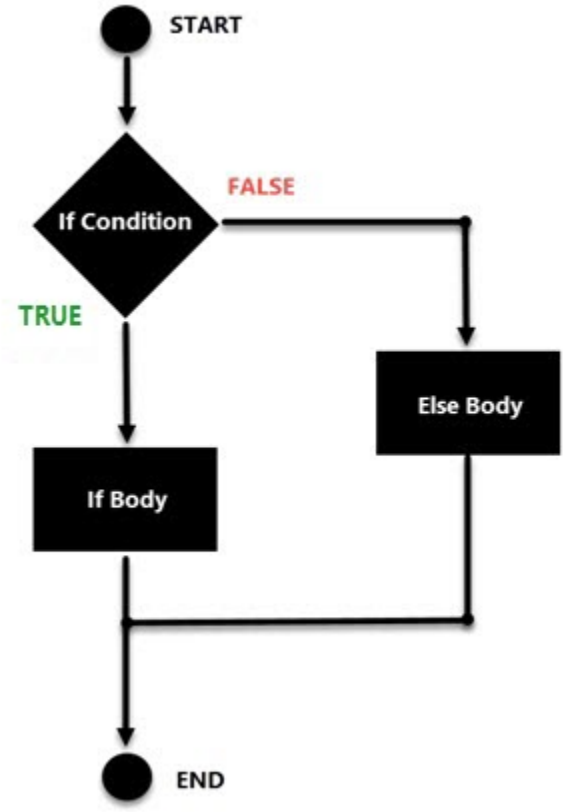
```
$var = 12

if($var -gt 0){
    'Positive'
}

if($var -lt 0){
    'Negative'
}

if($var -eq 0){
    'Zero'
}
```



Else Statement

Often you will want a piece of code to execute in your program when the condition in `if` statement was `False`, for this programming languages like of Python and PowerShell have a `else` statement. Or in other words this is fallback option if in case `if` statement fails.

An else statement can be combined with an `if` statement and is totally optional, but you can use it only once, which contains a code block that executes if and only if the conditional expression in the `if` statement was evaluated to boolean `False`.

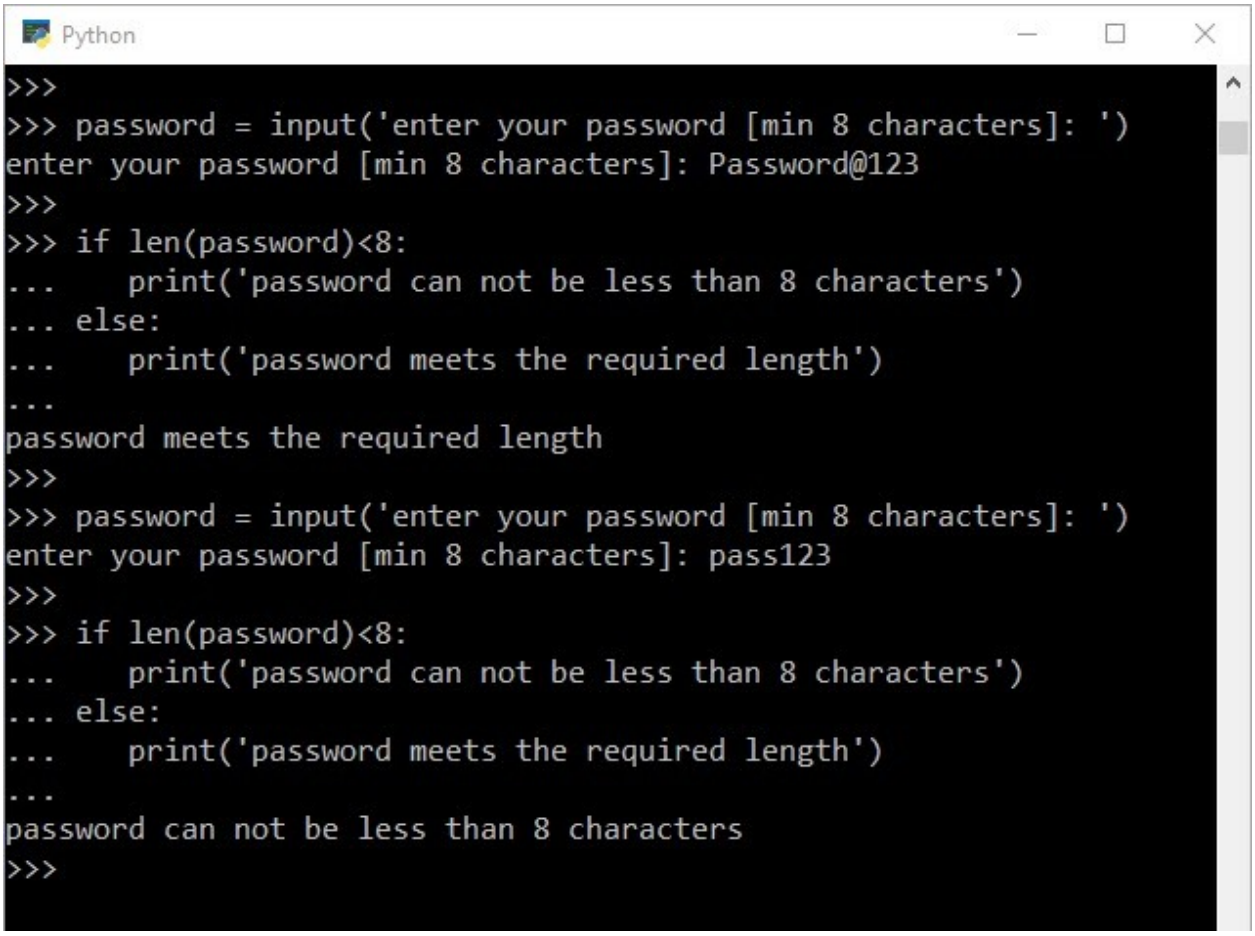


Syntax of `if..else` in Python:

```
if expression:
   statement
else:
   statement
```

We will take an example in Python to test the permitted length of password as a user input. Here we will use `if` statement to define a condition to test the Length and in case condition is evaluated to `False` the flow of control will jump to body of `else` statement and execute it.

```
password = input('enter your password [min 8 characters]: ')

if len(password)<8:
   print('password can not be less than 8 characters')
else:
   print('password meets the required length')
```

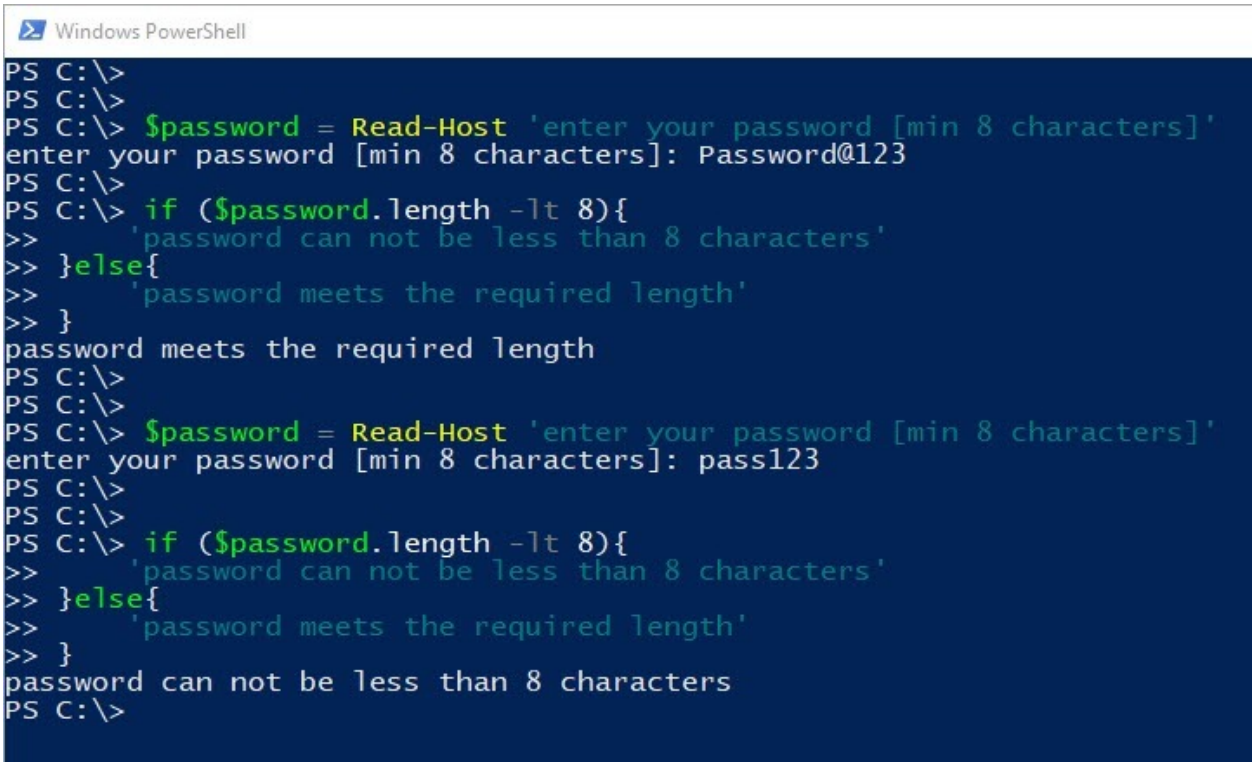


Syntax of `if..else` in PowerShell:

```
if(expression){
    statement
}
else{
    statement
}
```

Following code sample demonstrates a similar implementation of password length validation in PowerShell:

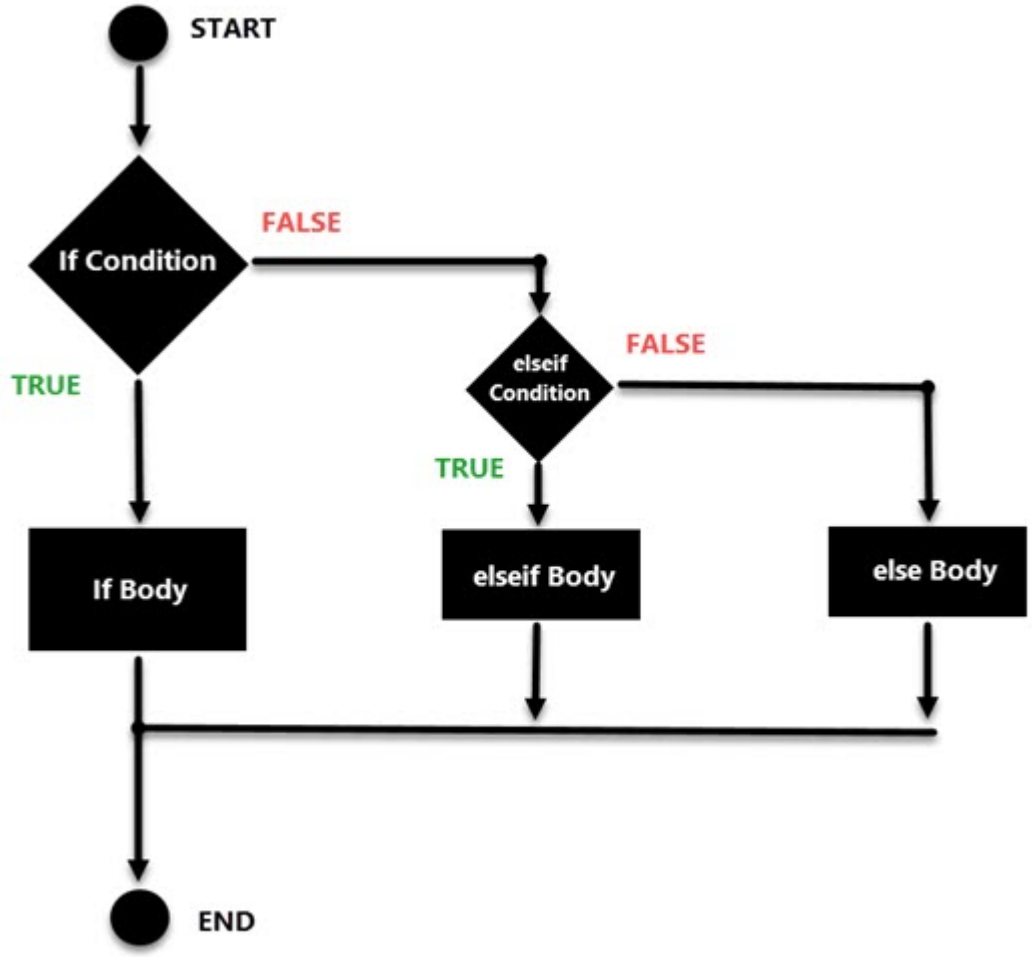
```
$password = Read-Host 'enter your password [min 8 characters]'

if ($password.length -lt 8){
    'password can not be less than 8 characters'
}else{
    'password meets the required length'
}
```



The elif and ifelse statement

So far with our conditional statements, we had an `if` statement and an `else` statement, which limits us to only two options, with each if statement evaluating to either `True` or `False` outcomes. But there are lot of scenarios where it is required to check more than two possible outcomes. To overcome this challenge programming languages offer a solution as an `elseif` statement like in PowerShell, which is used in Python as `elif`.

These statements are just like an `if` statement, and the whole purpose is to evaluate another condition. You can write as many `elseif` or `elif` statements that are required in the program but there can be only one `if` and one `else` statement.



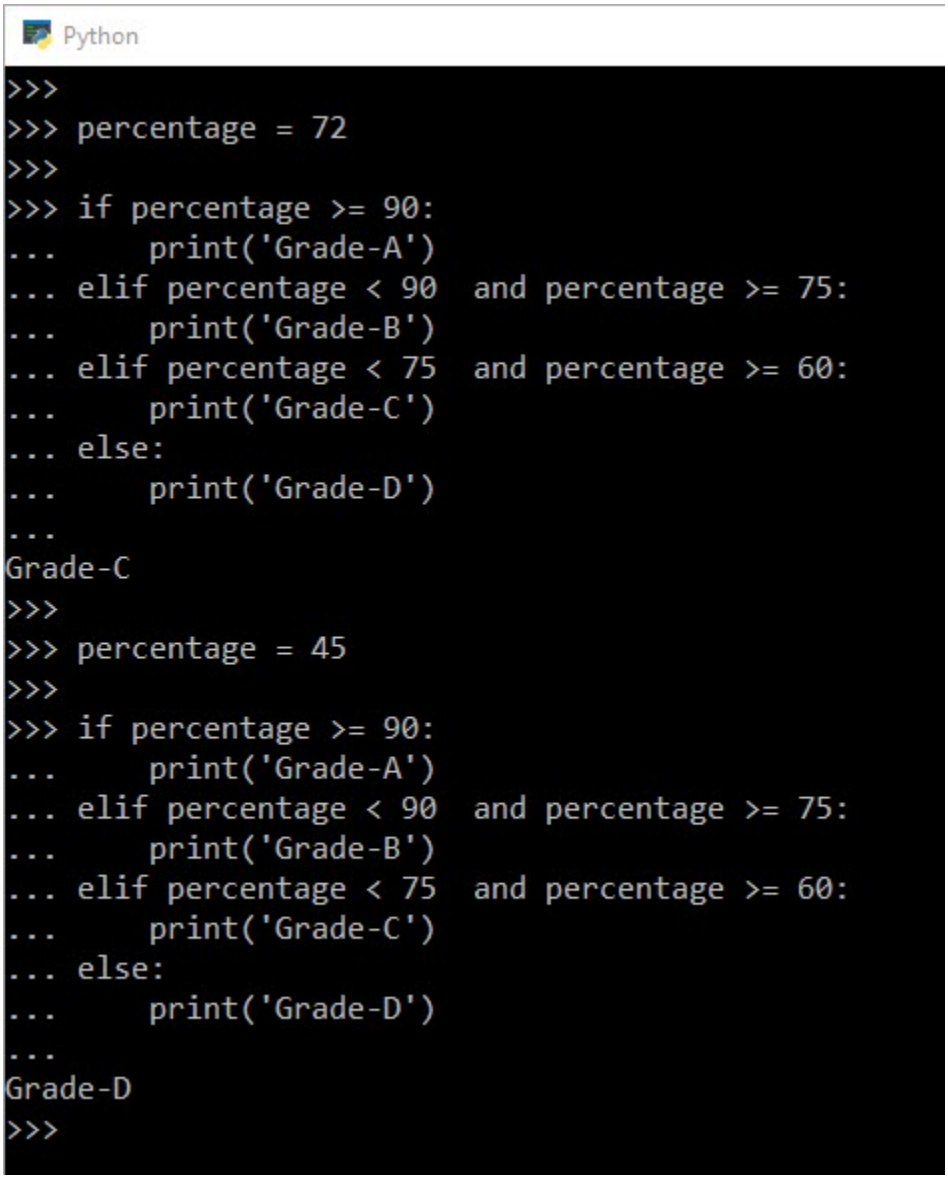
Syntax of `if..elif..else` in Python:

```
if expression1:
    statement
elif expression2:
    statement
elif expression3:
    statement
else:
    statement
```

Lets take a simple example to determine the grade of a student, in which we will define multiple conditions using `elseif` and `elif` conditional statements in PowerShell and Python respectively to check the range in which the student's percentage falls in:

```
percentage = 72

if percentage >= 90:
    print('Grade-A')
elif percentage < 90  and percentage >= 75:
    print('Grade-B')
elif percentage < 75  and percentage >= 60:
    print('Grade-C')
else:
    print('Grade-D')
```

In the above Python code sample, we used the `elif` statements to define more than one conditions to evaluate and determine the grade of the student.



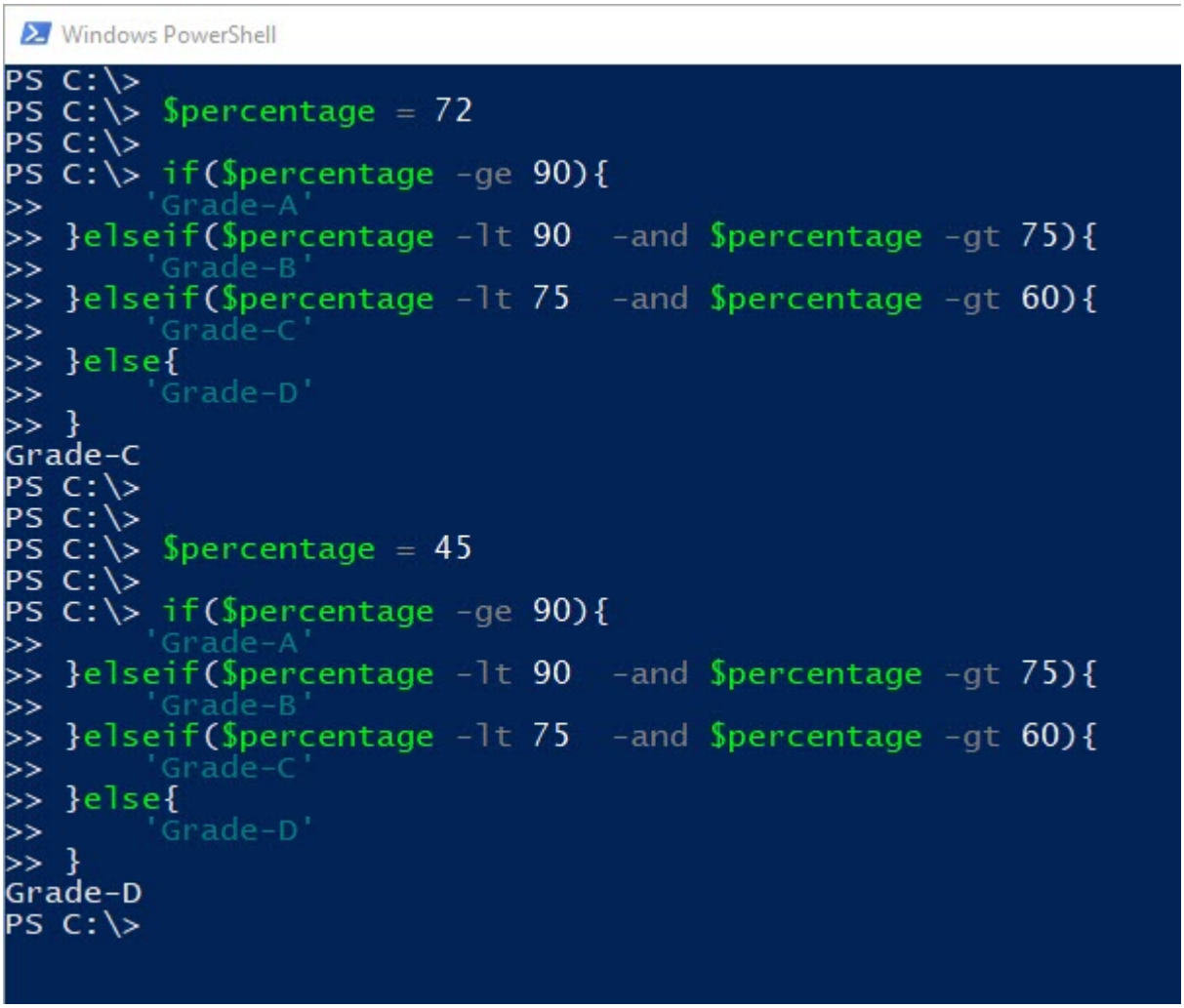
Whereas, in PowerShell the `elif` statement is written as `elseif` and we use parenthesis to define the conditions.

Syntax of `if..elseif..else` in PowerShell:

```
if(expression){
    statement
}
elseif(expression){
    statement
}
elseif(expression){
    statement
}
else{
    statement
}
```

Following is the same example implemented in PowerShell, but instead of using the Python's `elif` statement, we are using an `elseif` statement to define multiple conditions in our program.

```
$percentage = 72

if($percentage -ge 90){
    'Grade-A'
}elseif($percentage -lt 90  -and $percentage -gt 75){
    'Grade-B'
}elseif($percentage -lt 75  -and $percentage -gt 60){
    'Grade-C'
}else{
    'Grade-D'
}
```



Nested Conditional Statements

We have already covered the `if`, `elif`, `elseif` and `else` statements, but there are situations where we want to check for a secondary condition if the first condition is evaluated as `True`. For such cases we can define an `if..else` statement inside of another `if..else` statement, this is also called nesting of conditional statements.

Python Syntax:

```
if statement:
    print("outer if statement")
    if nested_statement:
        print("nested if statement")
    else:
        print("nested else statement")
else:
    print("outer else statement")
```

Let's again take the example of determining the student grade, but now we would be nesting conditional statements to check a secondary condition that is a passing or failing grade.

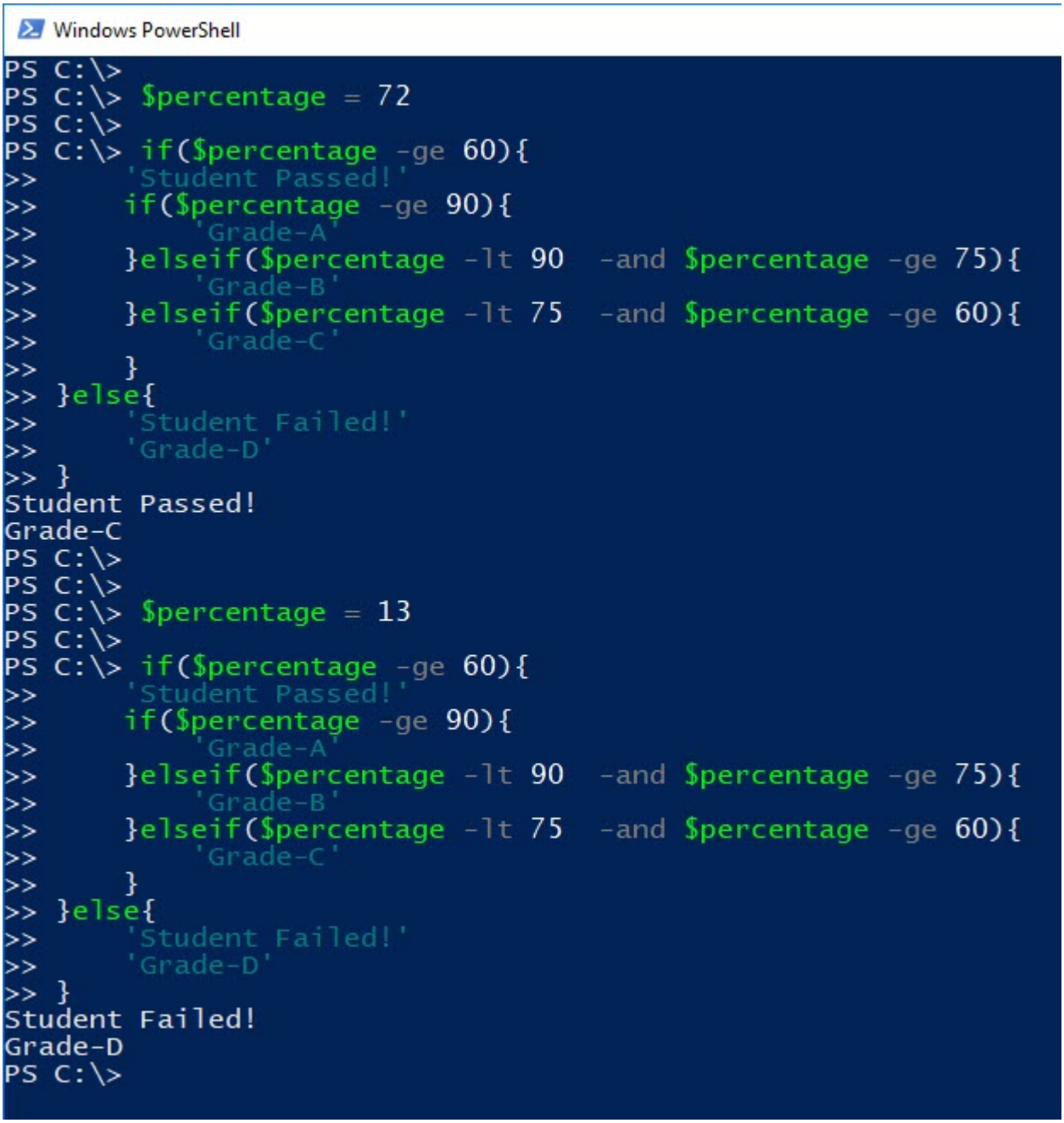
```
percentage = 72

if percentage >= 60:
    print('Student Passed!')
    if percentage >= 90:
        print('Grade-A')
    elif percentage < 90  and percentage >= 75:
        print('Grade-B')
    elif percentage < 75  and percentage >= 60:
        print('Grade-C')
else:
    print('Student Failed!')
    print('Grade-D')
```

In the above example if the outer `if` statement evaluates to `True`, the inner `if`, `elif` or `else` statements are evaluated, otherwise the outer `else` statement is evaluated.

Similarly we can implement this example with some minor syntax changes in PowerShell:

```
$percentage = 72

if($percentage -ge 60){
    'Student Passed!'
    if($percentage -ge 90){
        'Grade-A'
    }elseif($percentage -lt 90  -and $percentage -ge 75){
        'Grade-B'
    }elseif($percentage -lt 75  -and $percentage -ge 60){
```

```
        'Grade-C'
    }
}else{
    'Student Failed!'
    'Grade-D'
}
```



Chapter 15 - Loops

Loops are important in any programming language as they are utilized to execute a code block repeatedly. So in this chapter, we are going to learn about syntax and implementation of various loops in Python and PowerShell.

For Loop

You will often come across with scenarios where you would need to repeat or iterate a piece of code over and over, without writing the code block multiple times. The `for` loop is the most simple type of the loop that is also most preferred choice by programmers over any other loop type, but there are various ways in which you can use the `for` loop:

Using Range() Method

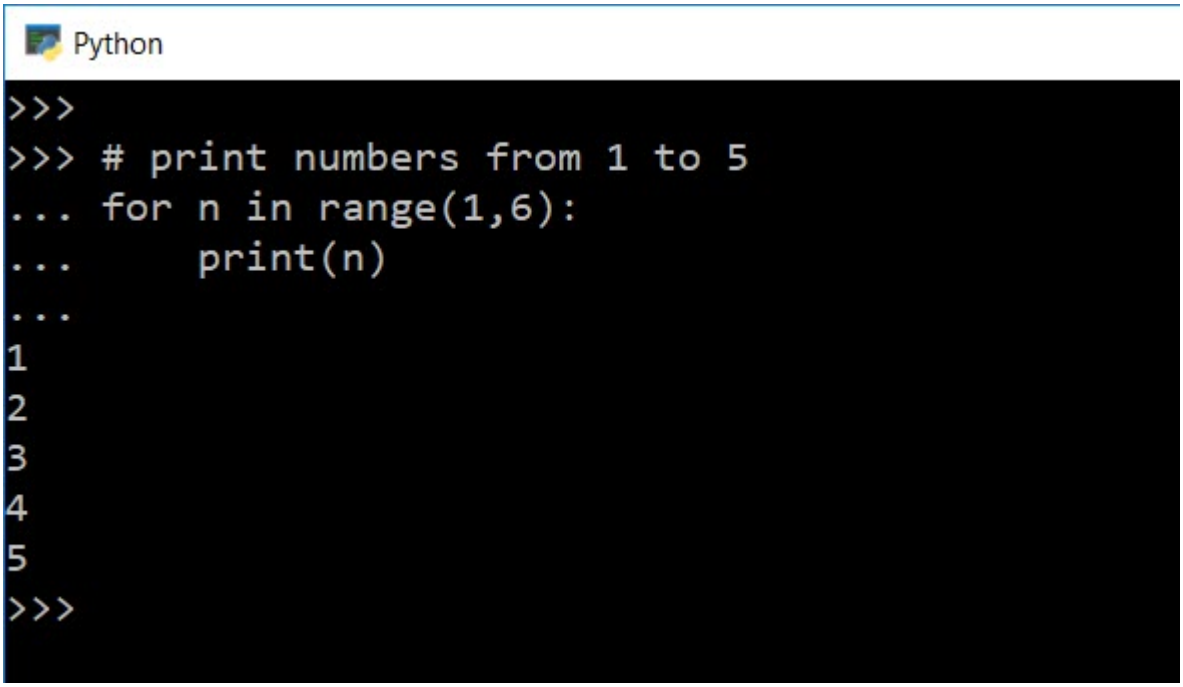
In Python, `for` loops can iterate over a number sequence using the `range()` function. The range function returns a list of numbers within a range that is specified as the function parameter(s).

Python Syntax for `for` loop:

```
for i in range(n):
    statement to iterate
```

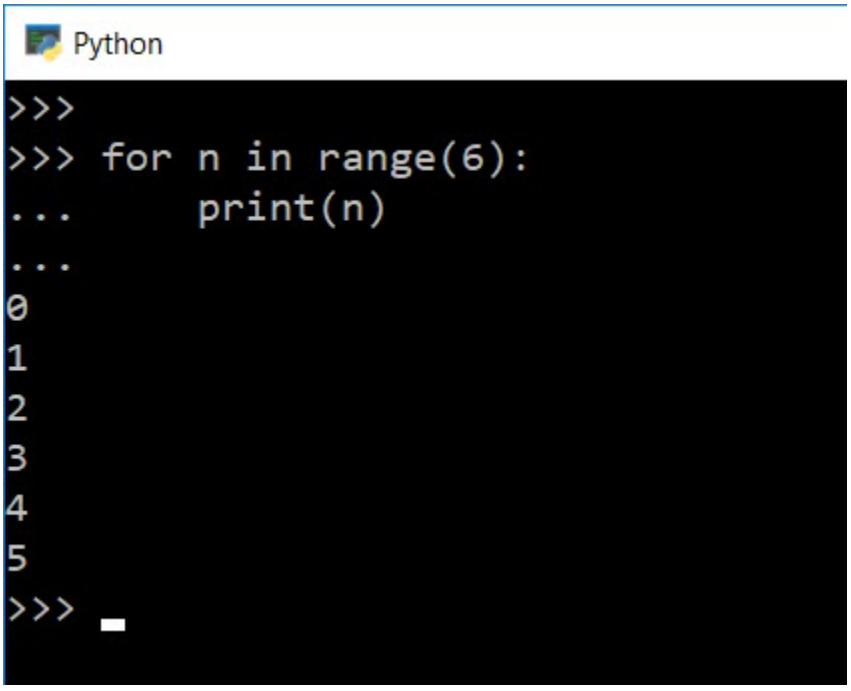
Let's take a simple example where we want to print a series of numbers: 1,2,3,4,5 using the `for` loop in Python

```
for n in range(1,6):
    print(n)
```



Note that the `range()` function is zero based that means the count starts from `0` and not from `1`, if the starting number is not defined. That means, in the above example, the result will be: `0,1,2,3,4,5` and not `1,2,3,4,5` if `range(6)` was used.

What is happening here? Let's try to understand by reading the above example from left to right: for each number in the range `1` to `5`, iterate the statements in body of the loop which is '`print(n)`' in our example.



Using List

Just like range of numbers you can also use elements of a Python List to iterate statements using a `for` loop, let me use the same example of printing numbers but with a list instead:

```
for n in [1,2,3,4,5,6]:
    print(n)
```

Good thing about Lists is that you can even use characters, string and floating point numbers to iterate, on contrary to `range()` function that can only have numbers.

```
# using list of characters
for n in ['a','b','c']:
    print('char:',n)

for n in list('abc'):
    print('char:',n)

# using list of strings
for n in ['abc','def']:
    print('String:',n)

# using list of floating point numbers
```

```
13 for n in [1.4, 2.3, 4.5]:
14     print('float:',n)
```

```
Python
>>>
>>> for n in ['a','b','c']:
...     print('char:',n)
...
char: a
char: b
char: c
>>>
>>> for n in list('abc'):
...     print('char:',n)
...
char: a
char: b
char: c
>>> for n in ['abc','def']:
...     print('String:',n)
...
String: abc
String: def
>>>
>>> for n in [1.4, 2.3, 4.5]:
...     print('float:',n)
...
float: 1.4
float: 2.3
float: 4.5
>>>
```

ForEach and ForeEach-Object

The PowerShell alternative of `for` loop is a `ForEach` statement and just like Python you can define a range of items to iterate and accessible within the loop body. But in PowerShell the double-dot operator (`..`) is used to define a range, that means `1..5` will return integers from 1 to 5 and -3..3 will return all the integers from negative 3 to positive 3 including zero.

PowerShell Syntax for `ForEach` loop:

```
Foreach($i in start..end){
    statement to iterate
}
```

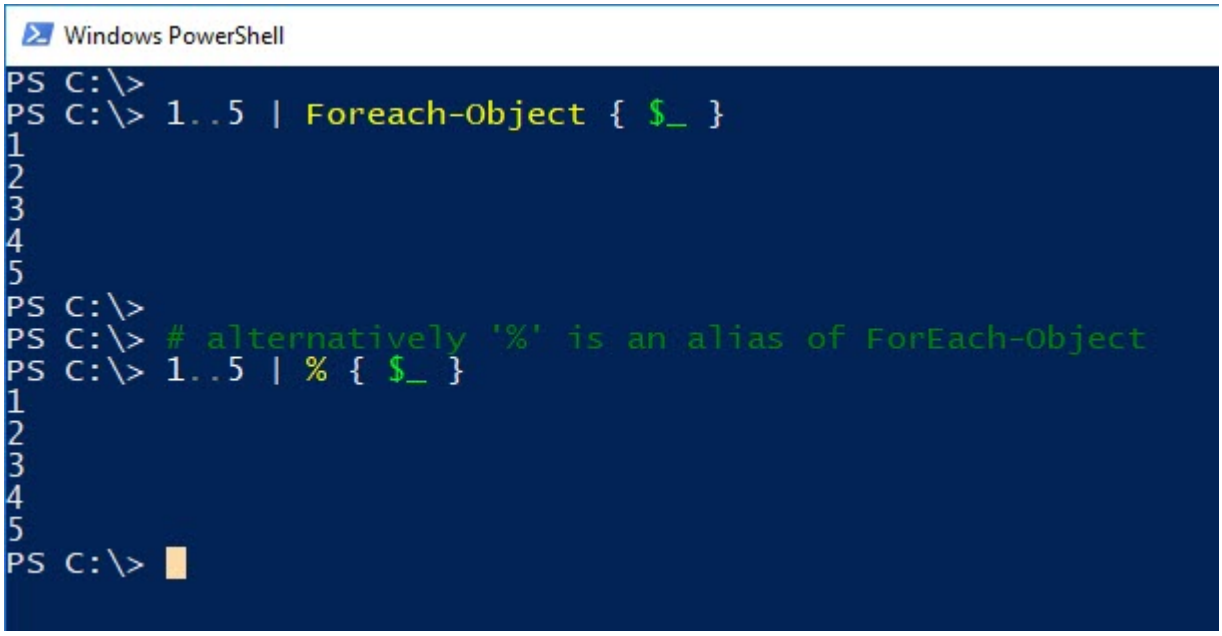
In the above syntax you have to provide `start` of the range and `end` of the range, and then the body of the loop with the statements to be iterated is enclosed inside braces `{ }`. Let's take the simple example of printing a series of numbers from 1 to 5 to demonstrate this:

```
Foreach($i in 1..5){
    $i
}
```

On other hand PowerShell has an optional approach to iterate items in a loop using the `ForEach-Object` cmdlet, but this cmdlet can only receive items from a PowerShell Pipeline like in the following example and mostly serve different use cases:

```
1..5 | Foreach-Object { $_ }

# alternatively '%' is an alias of ForEach-Object
1..5 | % { $_ }
```

Pipelines are most powerful feature of PowerShell that are used to connect series of commands together, which is also known as command chaining, and to know more about pipelines run: `Get-Help about_Pipelines` from the PowerShell console to read the detailed help description. Pipeline act as interface between output of the command on the left hand side and feed it as input to the command on right hand side, and in the above example `$_` represents the current object that was passed from left to right for processing. The beauty of pipelines are that objects can be passed from left to right one at a time as and when they are processed, so that command on the right doesn't have to wait for the complete command to be executed on the left and then pipe the results which makes them efficient in most use cases.



While Loop

The `while` loop also repeats the portion of code over and over for n-times, but if and only if a condition is met or is evaluated to `True`.

While Syntax in Python

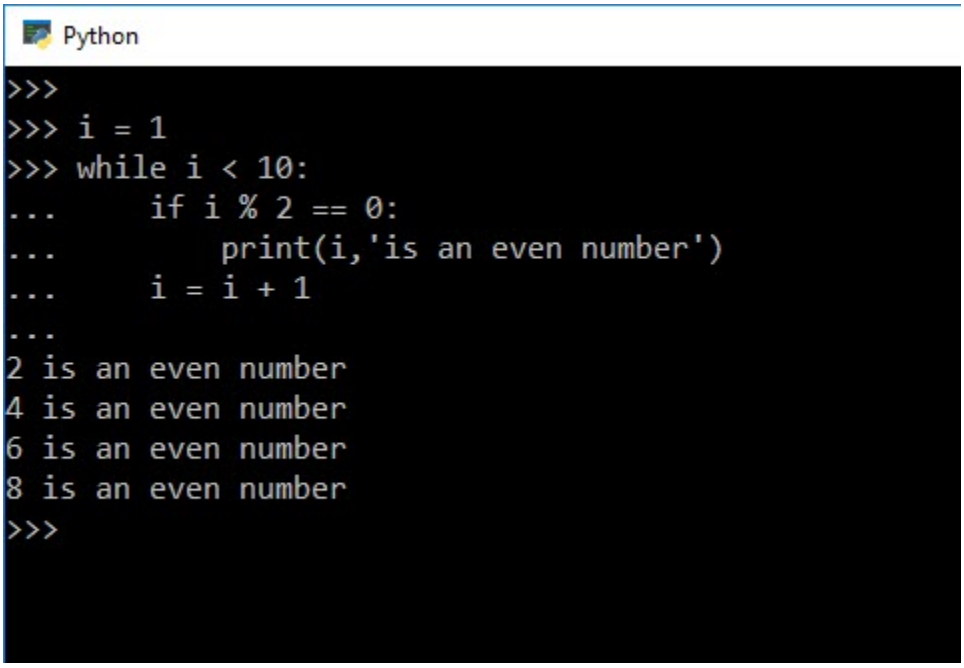
```
while condition:
    statement1
    statement2
```

While Syntax in PowerShell

```
while (condition) {
    statement1
    statement2
}
```

A simple example of `while` loop in Python can be only printing numbers from 1 to 10 that are even.

```
1  i = 1 # counter
2  while i < 10:
3      if i%2 == 0: # condition to check even number
4          print(i,'is an even number')
5      i=i+1 # increment the counter at the end of loop
```



In the above example first we define a counter variable at the top of the program and then check the value of the counter at the entry point of the `while` loop to make sure it is less than 10. If the condition evaluates to `True` then the body of the loop executes, where in we can again put a condition using an `if` statement to check if a number is divisible by 2 or not.

Please note that It is very important to increment the counter by one (Line-5) in the body of the `while` loop, otherwise the value of variable:`i` will always be 1 which is less than 10 and the program would end up in an infinite loop, because the while condition will always evaluated to `True`. So it is always a good practice to `break` the flow control in a loop when it is no longer required and in the next subsection we are going to discuss more about such control statements.

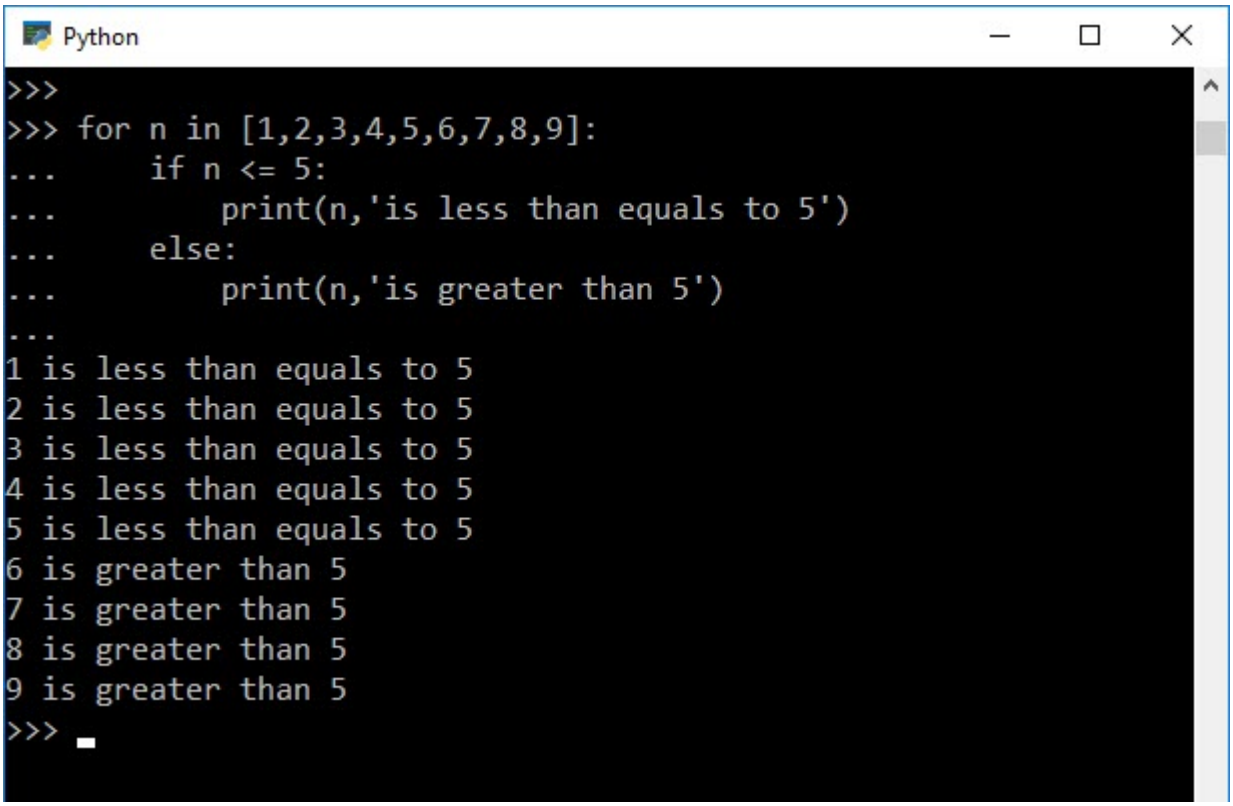
Loop Control Statements

All programming language has loop control statements that can change the flow of execution in a program from the normal sequence.

Continue

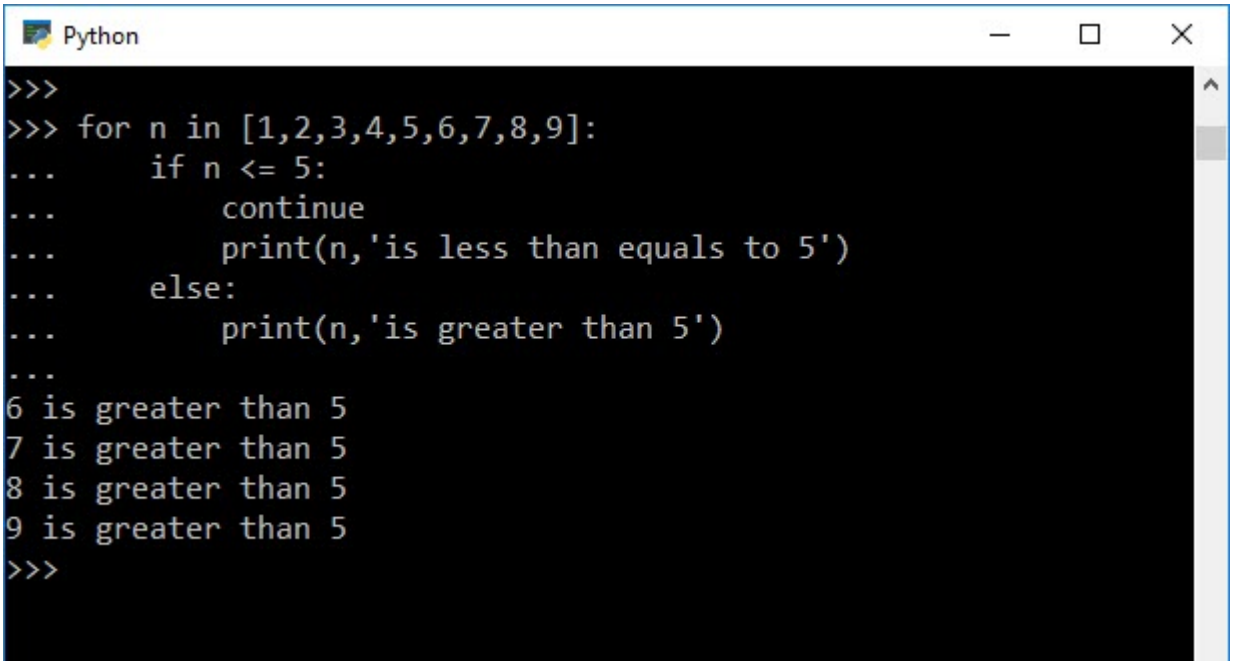
The `continue` statement returns the control to the beginning of the loop or in other words the flow control continues with the next iteration of the loop. In the following example the normal sequence of flow control should be: for each number in list, check if it is less than or equals to 5, then print it or else if it is greater than 5 print it with a different string.

```
1  for n in [1,2,3,4,5,6,7,8,9]:
2      if n <= 5:
3          print(n,'is less than equals to 5')
4      else:
5          print(n,'is greater than 5')
```



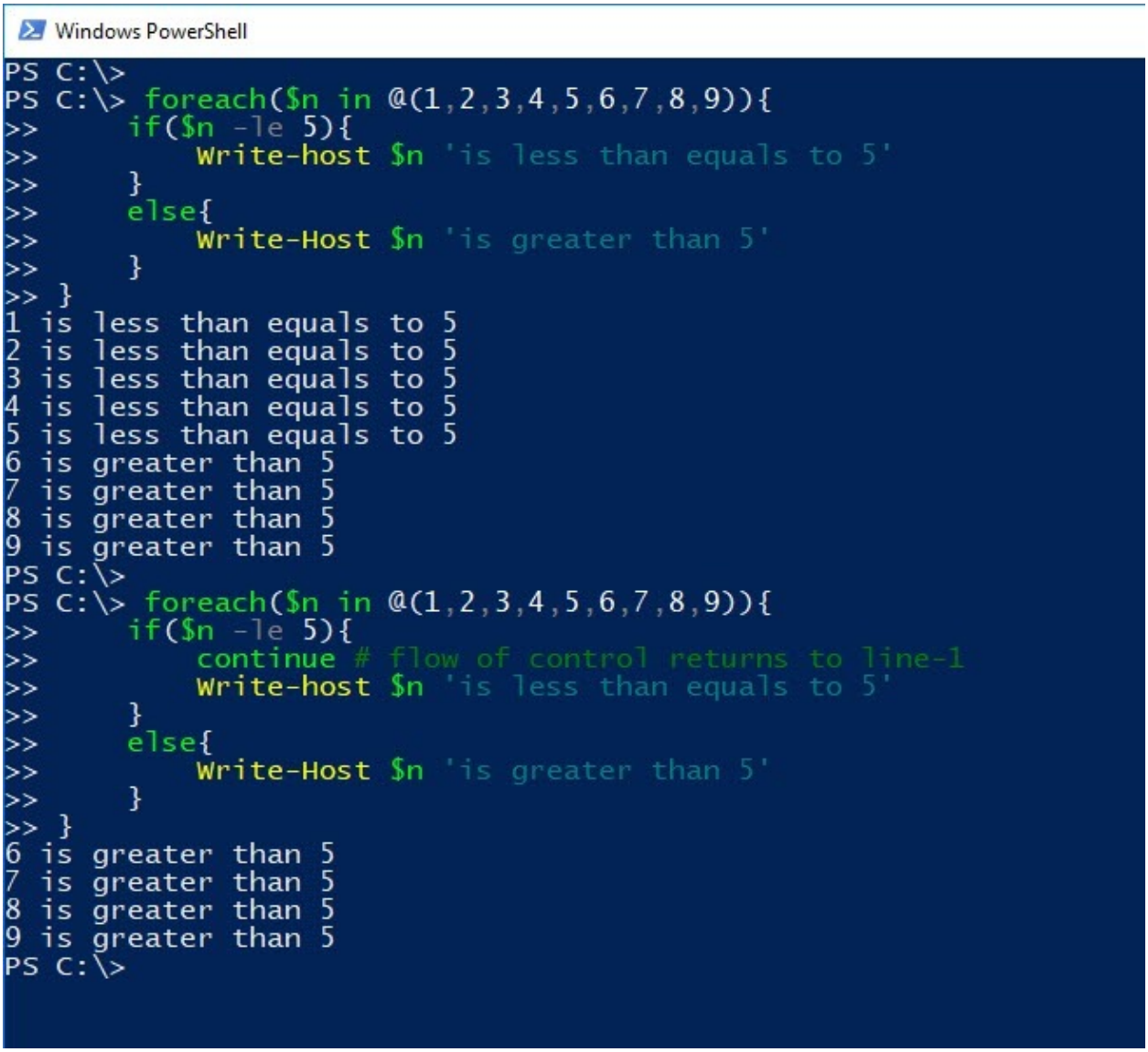
But imagine a use case when you want to print only the numbers greater than 5, then either you can use a `if` statement to filter only those numbers, or the second approach can be adding `continue` statement in our previous example to change the flow control to return to beginning of the loop if a number smaller than 5 is found. So that only numbers greater than 5 are returned.

```
for n in [1,2,3,4,5,6,7,8,9]:
    if n <= 5:
        continue # flow of control returns to line-1
        print(n,'is less than equals to 5')
    else:
        print(n,'is greater than 5')
```



Following is similar implementation on `continue` statement in PowerShell with almost no syntactical changes except the braces `{ }` and parenthesis `( )`.

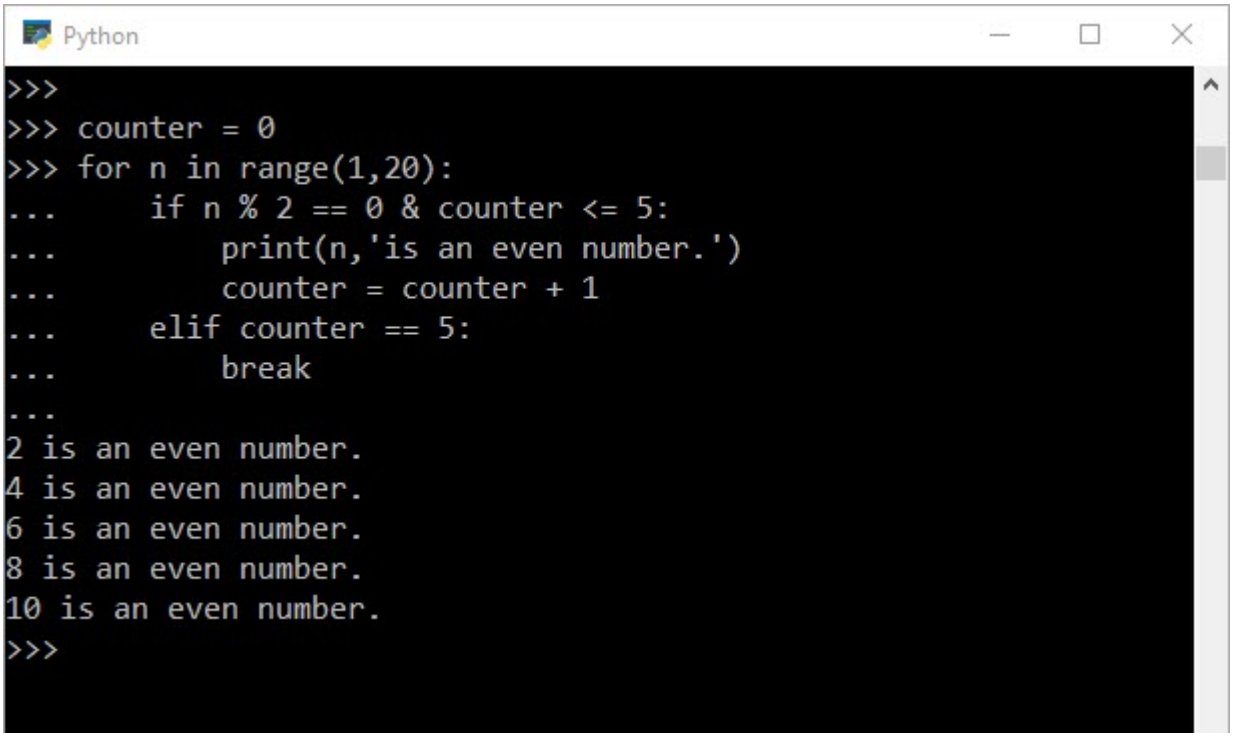
```
foreach($n in @(1,2,3,4,5,6,7,8,9)){
    if($n -le 5){
        continue # flow of control returns to line-1
        Write-host $n 'is less than equals to 5'
    }
    else{
        Write-Host $n 'is greater than 5'
    }
}
```



Break

The `break` statement brings control out of the loop. So let's take an example to make a Python program that can print first 5 even numbers from a range of numbers from 1 to 20:

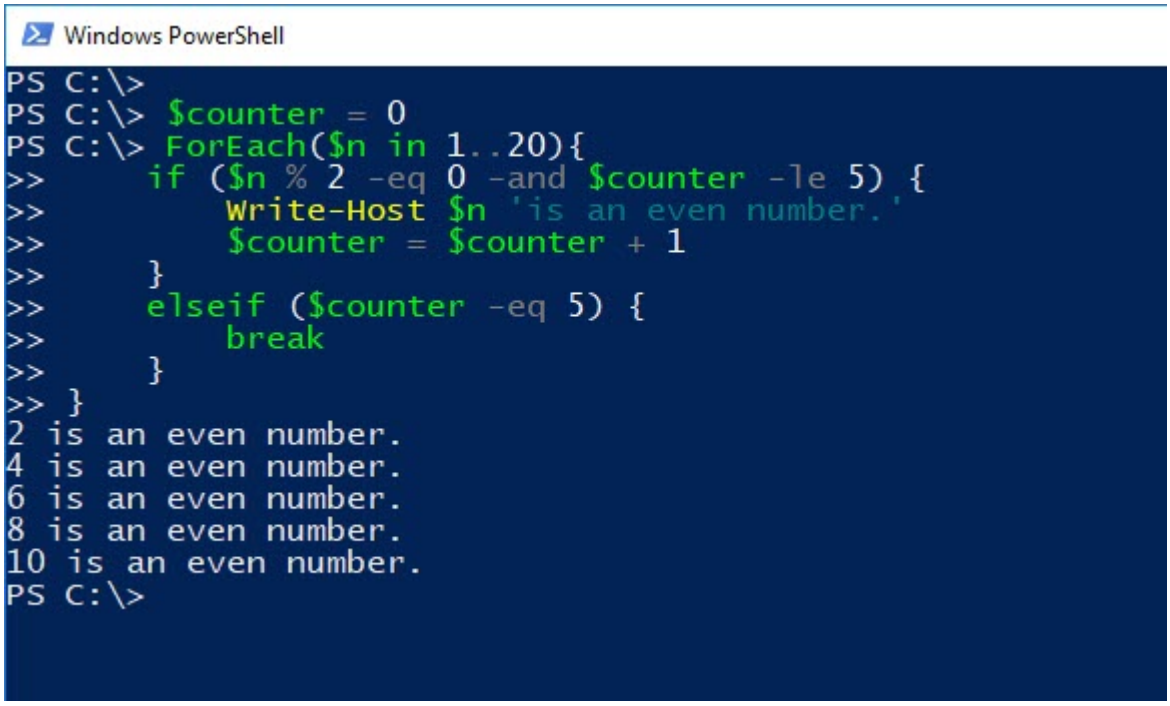
```
counter = 0
for n in range(1,20):
    if n % 2 == 0 & counter <= 5:
        print(n,'is an even number.')
        counter = counter + 1
    elif counter == 5:
        break
```



In the above program the we keep variable `counter` to keep count of even numbers found after each iteration of the loop from range 1 to 20. We also have an `if` statement with a condition to check if the number is even, and a secondary condition to make sure if the `counter` is equals to 5 then job of loop is done and flow control can come out of the loop with no more iteration required.

Similarly PowerShell can also use the `break` statement to bring the flow control out of the loop and jump to the next line following the loop in the program.

```
$counter = 0
ForEach($n in 1..20){
    if ($n % 2 -eq 0 -and $counter -le 5) {
        Write-Host $n 'is an even number.'
        $counter = $counter + 1
    }
    elseif ($counter -eq 5) {
        break
    }
}
```



Pass

The `pass` statement is primarily a placeholder for empty loops, conditional statements, functions and classes, so that Python code doesn't throw any error considering the syntactical need of the programming language, but performs no action.

```python
# press CTRL+C to interrupt
while True:
    pass

# empty class
class human:
    pass

# empty function
def test(a):
    pass
```

Nested Loops

Using one loop inside another loop is called nested loops in any programming language.

Syntax of nested `for` loop in Python:

```python
for i in range(n):
    for j in range(n):
        statement to iterate
```

Syntax of nested `while` loop in Python:

```python
while condition:
    while condition:
        statement to iterate
```

Let us take simple example of printing the following pattern using Python:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

In order to print this pattern you need to first understand that this pattern has `5` numbers and `5` rows to print, so we have to nest two loops like in the following code sample:

```
for i in range(1,6):
    for j in range(1,i+1):
        print(j, end=' ')
    print('')
```

A similar implemented of nested `for` loops in PowerShell:

```
ForEach($i in 1..5){
    Foreach($j in 1..$i){
        Write-Host "$j " -NoNewLine
    }
    Write-Host ""
}
```

Python

```
>>>
>>> for i in range(1,6):
...     for j in range(1,i+1):
...         print(j, end=' ')
...     print('')
...
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
>>>
```

Windows PowerShell

```
PS C:\>
PS C:\> ForEach($i in 1..5){
>>     Foreach($j in 1..$i){
>>         Write-Host "$j " -NoNewLine
>>     }
>>     Write-Host ""
>> }
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
PS C:\>
```

Please note that we can nest any type of loop inside of any other type of loop and we can even mix `for` loop inside a `while` loop or vice versa.

Chapter 16 - Functions

Functions are an essential part of all programming languages that provides a structure to your programs and make them modular and organized into blocks of reusable code which can perform a single or related functionality. The concepts of a function comes from mathematics where functions are used in computing one or more results depending upon the number of parameters passed to it. Use of functions is intended to increase the quality and comprehensibility of your program. So in this chapter we are going to dig deeper and understand how a function is created, called and used in Python and PowerShell programs.

Types of Functions

In this book you must have already come across some useful built-in functions, but there is more to functions and you can basically classify functions in following two categories:

1. **Built-in functions** - Functions that are built into the programming language like `help()` and `dir()` in Python and `more` and `prompt` in PowerShell.
2. **User-defined functions** - Functions that are defined by the users are called the user defined functions or custom functions, and gives the programmer ability to extend beyond the built-in functions of the programming language.

So in this chapter we will mainly cover the user defined functions.

Creating a Function

Before you can even use a function you have to first define it with a name, a set of parameters and the code block this is also called the function definition, both Python and PowerShell follow a different syntax and keywords which are following:

Python Syntax:

```
def function_name (param1, param2):
    "function docstring"
    function statements
    return <expression>
```

PowerShell Syntax:

```
function function_name ($param1, $param2){
    <#
    .SYNOPSIS
    function help documentation
    #>
    function statements
    return <expression>
}
```

By default, both in Python and PowerShell parameters are positional and arguments will bind in the same order that they were defined. Following are couple of rules to keep in mind when defining a function:

- Python function block begins with the keyword `def` followed by function name and parenthesis `( )`, you can define the function parameters and arguments in the parenthesis and end the line with a colon `:`

```
  def function_name (param1, param2):
```

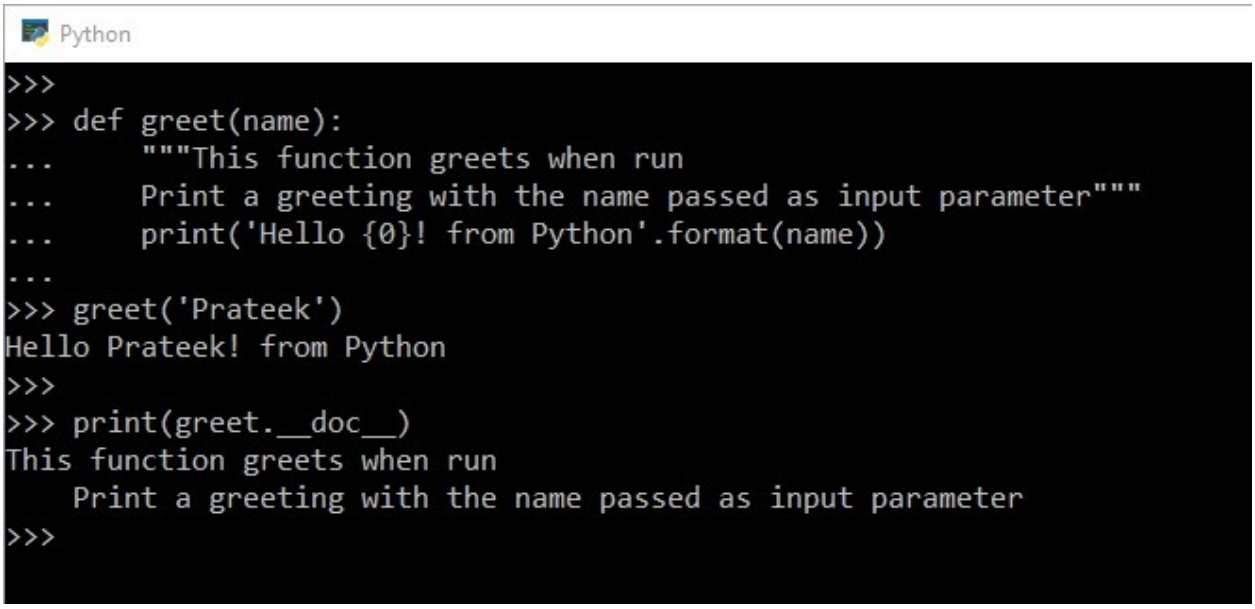
  On the other hand, PowerShell function block begins with keyword `function` and you end the line by opening a brace `{`

```
  function function_name (param1, param2) {
```

- In Python the first statement of a function is optional statement also know as the documentation string of the function or `docstring`.

```
  def greet(name):
      """This function greets when run
      Print a greeting with the name passed as input parameter"""
      print('Hello {0}! from Python'.format(name))
```

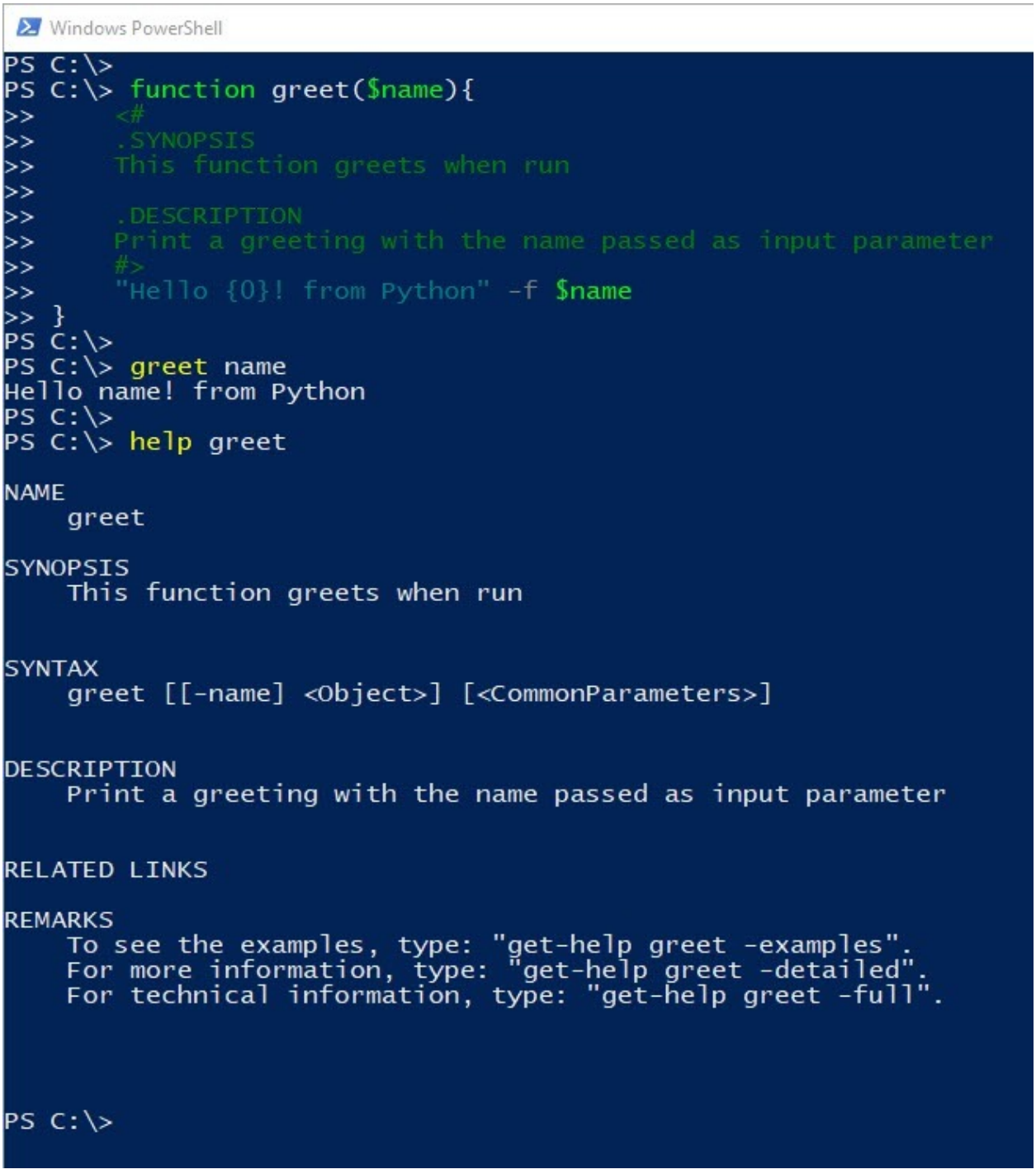
  and to access the `docstring` of a function, you've simply provide the function name followed by (.) Dot operator and `__doc__` property.



PowerShell doesn't support a `docstring` but instead you can add help information for documentation purpose using a multi-line comment, like in following code sample:

```
function greet($name){
    <#
    .SYNOPSIS
    This function greets when run

    .DESCRIPTION
    Print a greeting with the name passed as input parameter
    #>
    "Hello {0}! from Python" -f $name
}
```

To view help description of this function you simply use the `Get-Help <Function Name>`



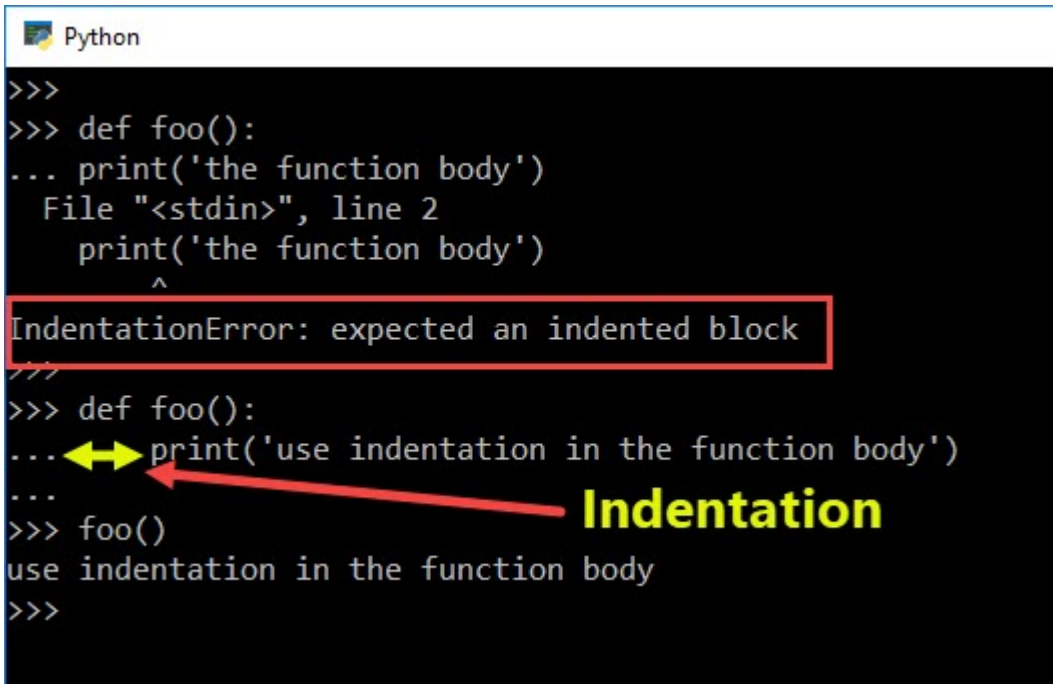
- After the `docstring` you can add all the statements that the functions is required to execute: Logical statements, loop etc in the body of the function.
- Indentation in Python is something to keep in mind while defining your functions. Python has a particular style of indentation to define the code, since Python functions don't explicitly begin or end using the curly braces to indicate the start and stop for the function body like in PowerShell, so Python relies on indentation for that. This means you have to add indentation

(space) in line-2 of the following code because that is how Python understands that it is beginning of the function body where you define all you statements that you want to execute when the function is called.

```
1   def foo():
2       print('use indentation in the function body')
```

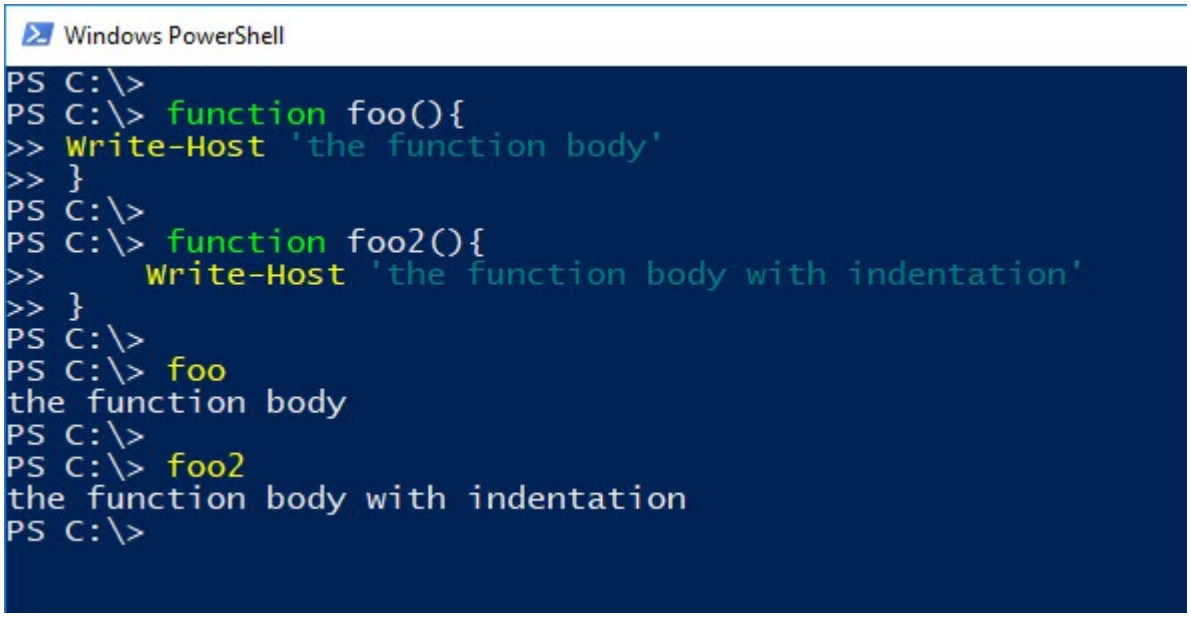
And if you don't ident your code properly you will run into indentation errors like in the following example:

```
1   def foo():
2   print('the function body')
```



Whereas, PowerShell as a programming language is very relaxed and you use curly braces to define the begin and end of the function body. Indentation or no indentation does not matter in PowerShell functions.

```
1   function foo(){
2   Write-Host 'the function body'
3   }
4
5   function foo2(){
6       Write-Host 'the function body with indentation'
7   }
```

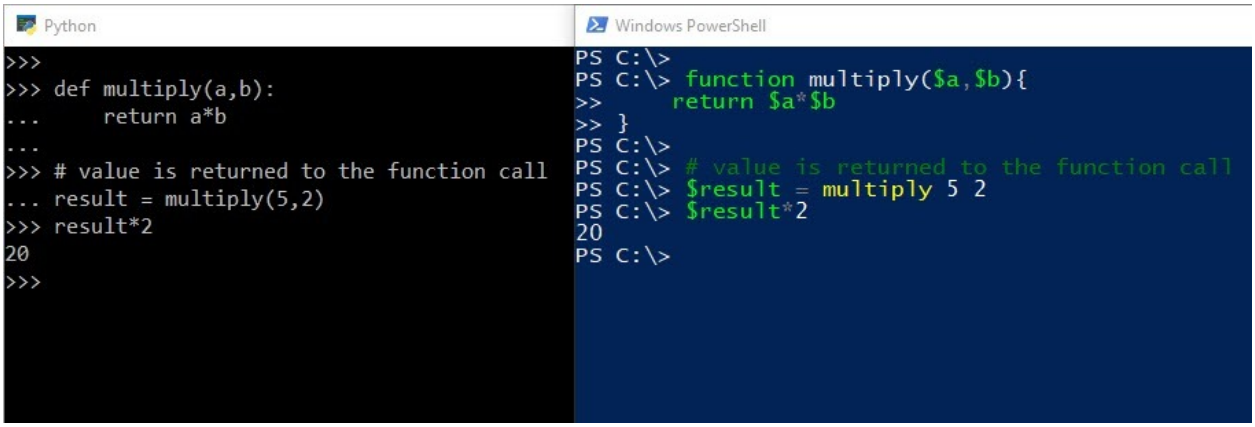


- In Python a function ends with a `return` statement if in case the function is returning an output, but without a return statement, your function will return an `None` object.

```
def multiply(a,b):
    return a*b

# value is returned to the function call
result = multiply(5,2)
result*2
```

Likewise you can also return values and flow control back to from where the function was called in PowerShell using the `return` statement

```
function multiply($a,$b){
    return $a*$b
}

# value is returned to the function call
$result = multiply 5 2
$result*2
```



Calling a Function

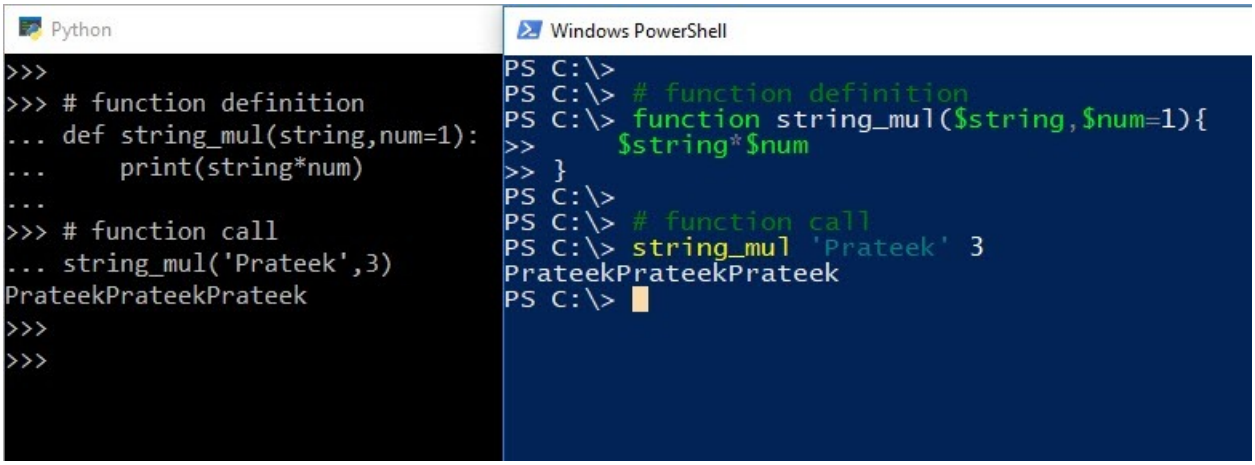
Function definition is just a structure and to execute the function directly from the Python console, program or even inside another function you have to call it. Basically calling a function is telling the program to execute the function. To call the function, you write the name of the function followed by parentheses and pass parameters to the function if it is required inside the parentheses.

Let's suppose you have a function that multiplies a string by the argument you supply to the parameters, then in Python you can simply call the function using the name of function followed by parentheses and arguments inside it:

```
# function definition
def string_mul(string,num=1):
    print(string*num)

# function call
string_mul('Prateek',3)
```

PowerShell function calls are little bit different as in you just write the name of the function followed by arguments separated by spaces:

```
# function definition
function string_mul($string,$num=1){
    $string*$num
}

# function call
string_mul 'Prateek' 3
```



Calling a Function

Functions vs Methods

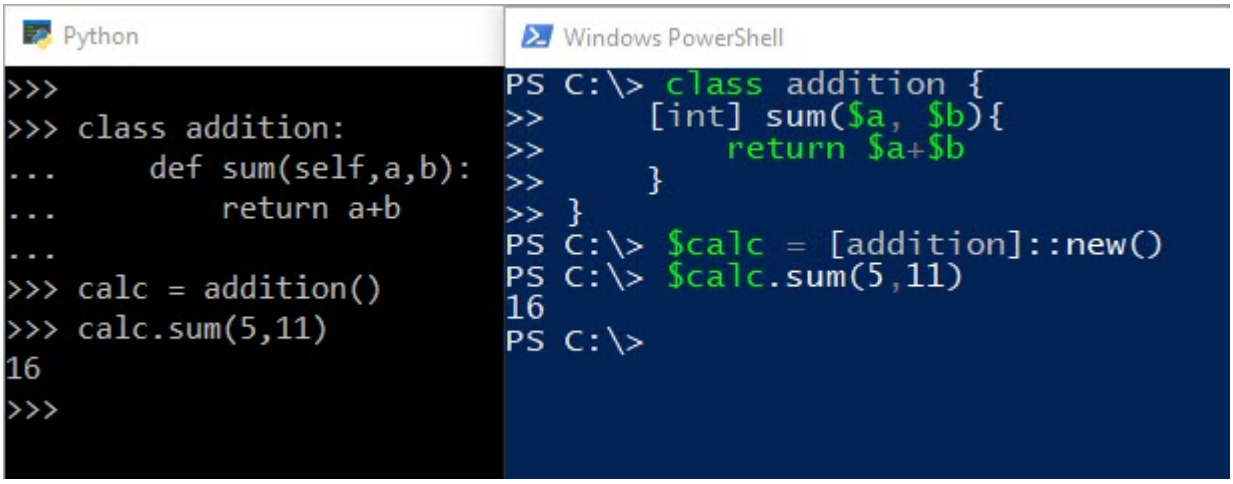
A common misconception or area of confusion is that: Method and Function are used interchangeably or as synonyms which is not at all correct. A method is referred to a function which is part of a class and defined inside the body of the class, and to access a method you have to access instance or object of the class using the (.) Dot operator.

Python:

```
# define a class in Python
class addition:
    # define a method inside a class
    def sum(self, a, b):
        return a+b
```

PowerShell:

```
# define a class in PowerShell
class addition {
    # define a method inside a class
    sum($a, $b){
        return $a+$b
    }
}
```



Methods in Python and PowerShell

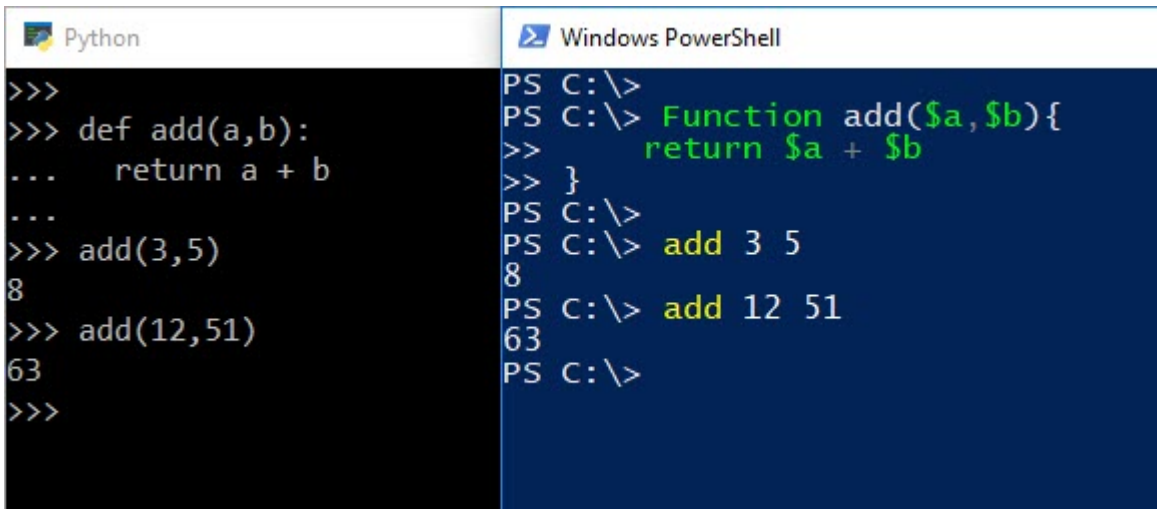
Whereas, a function is not necessarily have to be defined in a class and can exist as a standalone function. This implies that all methods are functions but not all functions are methods.

Python

```
# define a function
def add(a,b):
  return a + b
```

PowerShell:

```
Function add($a,$b){
    return $a + $b
}
```



Functions in Python and PowerShell

Parameters and Arguments

There is always a little bit confusion about parameters and arguments, let me begin by saying that they are not synonymous or same. Parameters are the names (of variable) while defining a function or a method, and the arguments are values mapped to these parameters. In other words, arguments are passed or supplied through the function call and is stored in parameters so that it is accessible in the function body.

Considering the following example `num` is the function parameter and integer `3` is the argument passed to the function when it was called and is accessible inside the function body

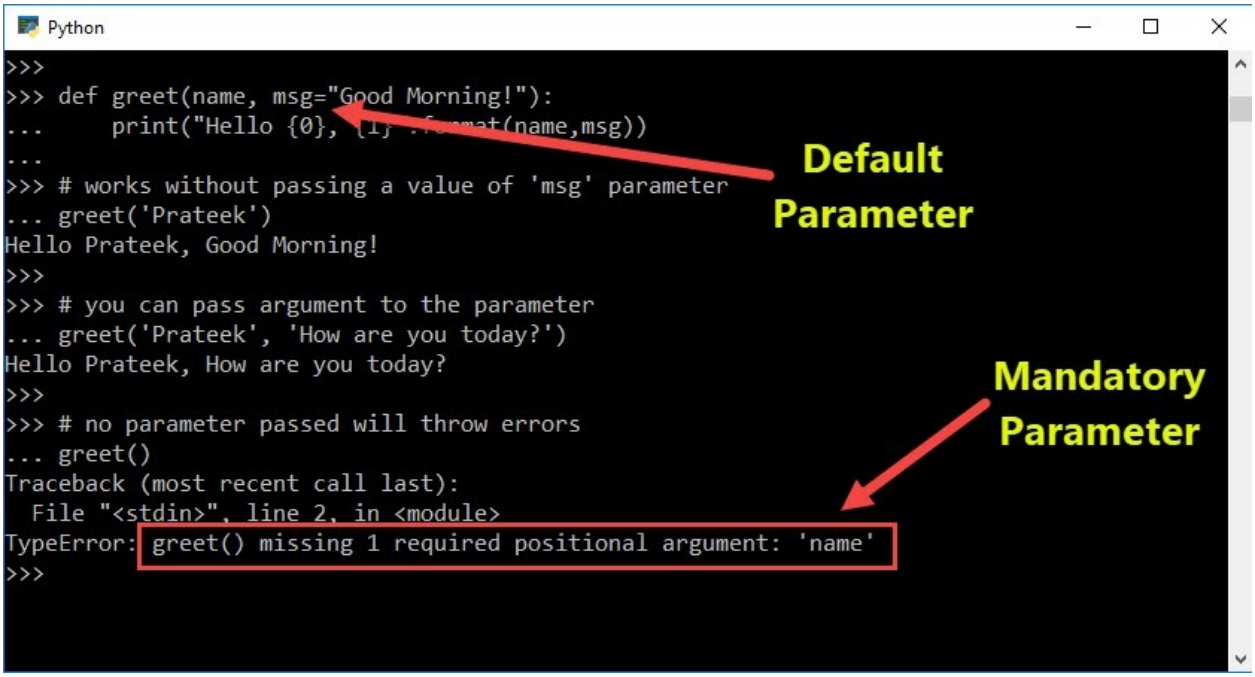
```
def double(num):
    return num*2

double(3)
```

Mandatory and Default parameters

In Python by default any parameter defined in a function is a mandatory parameter, and in case you miss them in function calls you will run into errors. But Python functions can also have optional parameters, which are also called default parameters. Default parameters are parameters that are not necessarily have to be supplied during the function call, this works because when you define optional parameters you provide it a default value to the parameter, so that when no value is supplied the default value is used. The following implementation of functions in Python demonstrates how to define a mandatory and a default parameter.

```
def greet(name, msg="Good Morning!"):
    print("Hello {0}, {1}".format(name,msg))

# works without passing a value of 'msg' parameter
greet('Prateek')

# you can pass argument to the parameter
greet('Prateek', 'How are you today?')

# no parameter passed will throw errors
greet()
```

In the above function:`greet` the `name` is the mandatory parameter and `msg` is the default parameter with a `String` value `Good Morning` assigned to it.

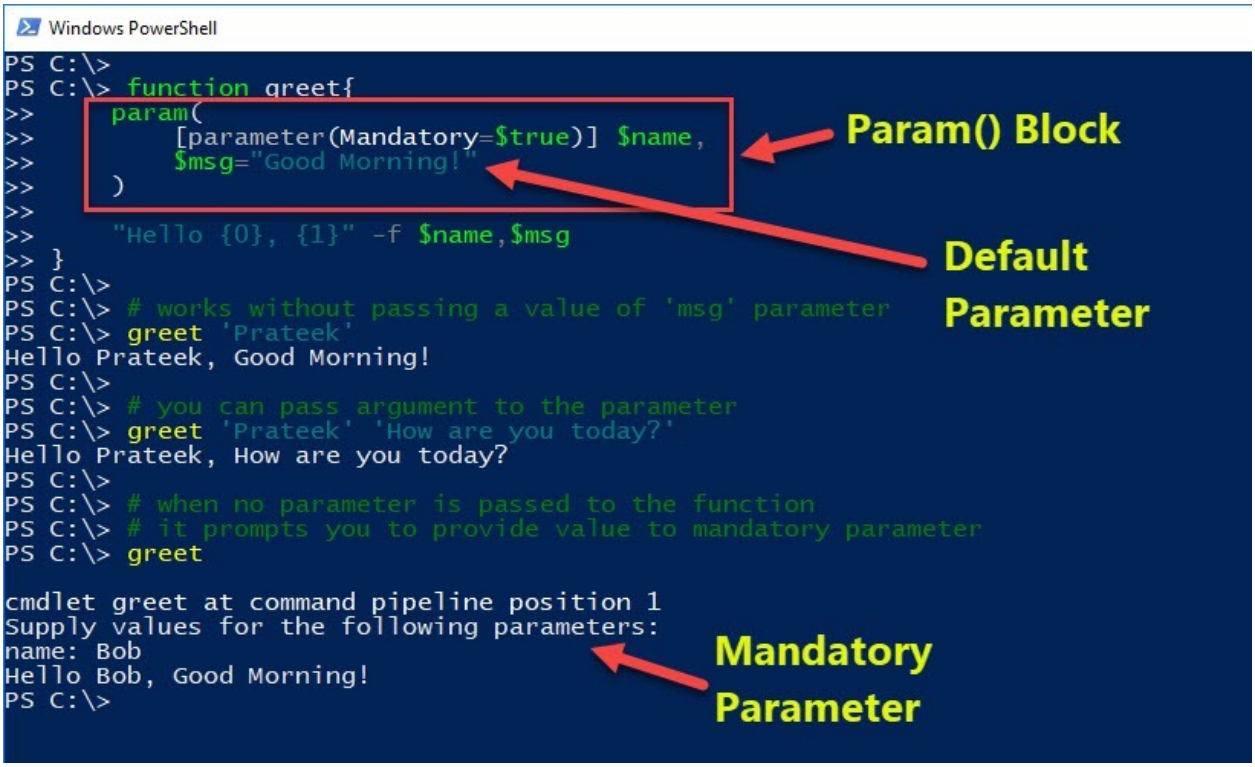


On other hand in PowerShell there are two ways to define parameters of a function, one is the standard way in which you define the parameters in a parentheses just after the function name, but there is one more approach which is a bit non-traditional way to define parameters using a `param()` block. To know more about `param()` block run: `Get-Help About_Functions_Advanced_Parameters` from your PowerShell console.

Inside the `param()` block you can define parameters and their attributes like when you declare `[parameter(Mandatory=$true)] $name` it makes the `name` a mandatory parameter of the function. Similarly to define a parameter with a default value you use the assignment operator `$msg="Good Morning!"` to provide the default argument to the parameter in case it is not supplied when calling the function.

```
function greet{
    param(
        [parameter(Mandatory=$true)] $name,
        $msg="Good Morning!"
    )

    "Hello {0}, {1}" -f $name,$msg
}

# works without passing a value of 'msg' parameter
greet 'Prateek'

# you can pass argument to the parameter
```

```
14 greet 'Prateek' 'How are you today?'
15
16 # when no parameter is passed to the function
17 # it prompts you to provide value to mandatory parameter
18 greet
```



The Anonymous Functions

Functions without a name are called anonymous functions, because they are not declared by the standard approach by using the `def` keyword in Python or `function` keyword in PowerShell. In Python you can use the `lambda` keyword to create small anonymous functions that can have multiple arguments and an expression that is evaluated in a single statement on basis of the arguments passed.

Syntax of lambda function in Python:

```
1 lambda [arg1 [,arg2,.....argn]]: expression
```

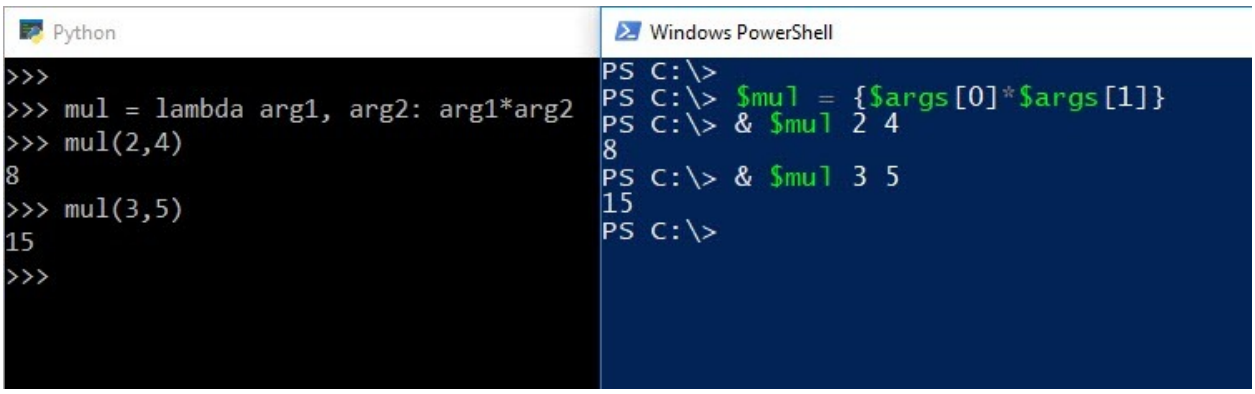
```
# defining a lambda function
mul = lambda arg1, arg2: arg1*arg2

# calling a lambda function
mul(2,4)
mul(3,5)
```

A similar implementation in PowerShell can be achieved by defining a script block which can access the arguments passed to the script block using the automatic variable `$args` and can evaluate an expression depending upon arguments supplied during the function call. To know more about Script blocks and automatic variable read the help documentation of these in PowerShell:

```
Get-Help About_Script_Blocks
Get-Help About_Automatic_Variables
```

Assign the script block to a variable, and now this variable can be treated as a PowerShell function or in other words you can call them anonymous function, because it doesn't have a name by which it can be referred or called. In order to call this anonymous function you have to use the call operator (`&`) followed by the variable name that stores the script block and you can also pass the arguments to the parameters separated by spaces like in the following code sample which will evaluate the expression inside the code block and return results:

```
# defining a anonymous function
$mul = {$args[0]*$args[1]}

# calling a anonymous function
& $mul 2 4
& $mul 3 5
```



Anonymous Functions

Chapter 17 - Handling XML and JSON Data

XML has been the go-to format and the only choice of programmers for open data interchange for a long period of time. But over recent years JSON has also gained some popularity due to its lightweight nature and faster parsing. Apart from the differences between them, they both have been adopted by all major programming languages of the world. So in this chapter, we are going to dig deeper how to read/parse, build XML and JSON formats and at last we will also try to understand how serialization and de-serialization work with these two formats.

Difference between JSON and XML

Generally speaking JSON is a lightweight cousin of XML and there is no clear superiority of one format against the other and your final choice should depend on what you really need and your use case.

`XML` stands for **Extensible Markup Language**, which is an open source language that is heavily used by programmers worldwide to store, persist and transport structured data, so that it can be read by other programs and software independent of underlying operating system, because XML data can be stored in a text file and can be shared or exchanged on any operating system. This is extremely useful to store small amount of data in a simple, human- /machine-readable structured format, which would otherwise require a SQL database.

XML Example:

```
<employees>
  <employee>
    <first>Prateek</first>
    <last>Singh</last>
  </employee>
  <employee>
    <first>Peter</first>
    <last>Parker</last>
  </employee>
</employees>
```

`JSON` stands for **JavaScript Object Notation**, which is a simple text-oriented format to store data and exchange information between programs, browsers(client) and servers. Slowly it has become

one of the most popular and preferable data exchange formats compared to XML in various use cases and is now support by most of the modern programming languages.

JSON Example:

```
{"employees":[
  { "first":"Prateek", "last":"Singh" },
  { "first":"Peter", "last":"Parker" },
]}
```

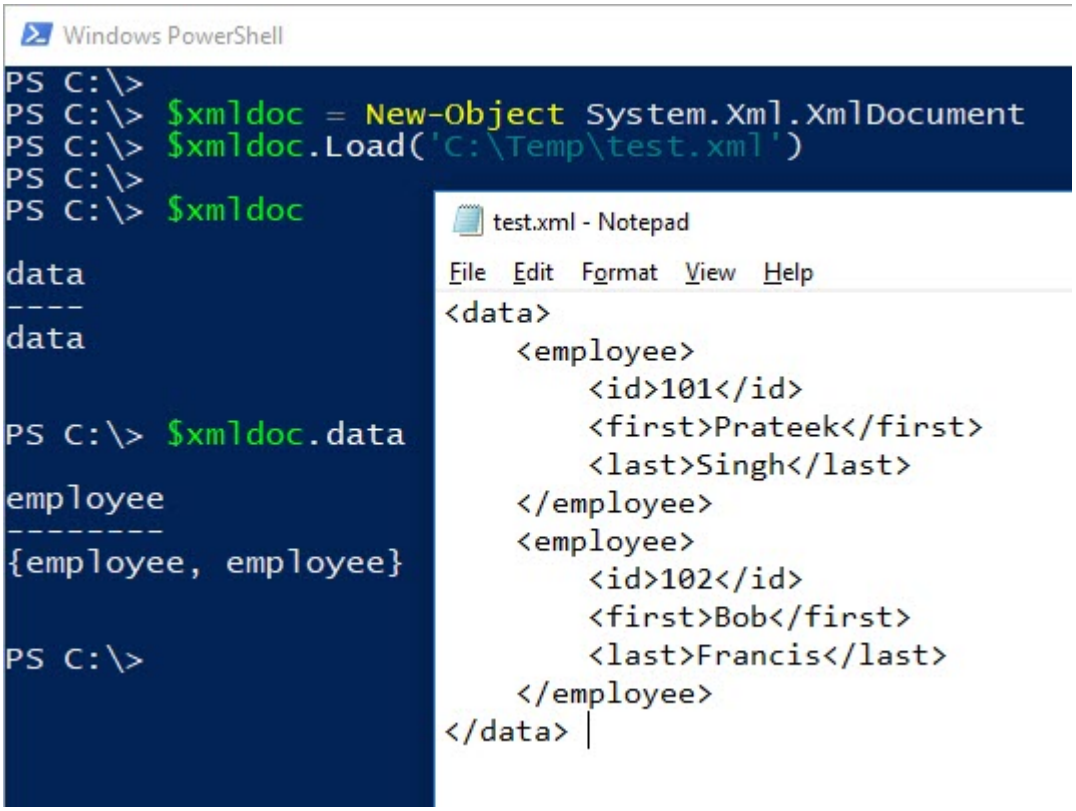
JSON	XML
Stands for JavaScript Object Notation	Stands for eXtensible Markup Language
Data oriented	Document oriented
Simple to read and write	Less simple than JSON
Less verbose, light Weight and faster to parse	Heavier than JSON
No support to display the content	Capability to display data and stylesheet transformations (XSLT)
Supports array	XML doesn't support array

Reading XML from a file

PowerShell has access to `.Net` class: `System.Xml.XmlDocument` which can be used to load a XML document and parse its content. All we have to do is instantiate this class using the `New-Object` cmdlet, then use the `Load()` method to load a XML file by passing a valid file path. Once the file is loaded it will parse the XML document and convert it to PowerShell objects, in our example we have a following XML sample saved as `test.xml` file, which stores the Employee data like: `ID`, `first` and `last` name into a structured format in child tags\nodes like `<id>..</id>`, `<first>..</first>` under parent tags\nodes like `<employee>..</employee>`.

```
<data>
    <employee>
        <id>101</id>
        <first>Prateek</first>
        <last>Singh</last>
    </employee>
    <employee>
        <id>102</id>
        <first>Bob</first>
        <last>Francis</last>
    </employee>
</data>
```

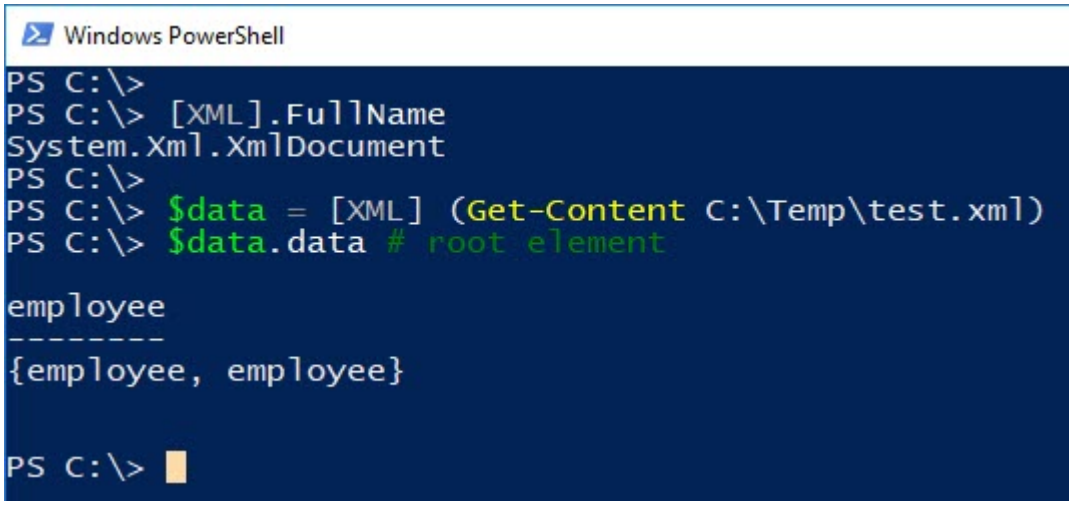
```
$xmldoc = New-Object System.Xml.XmlDocument
$xmldoc.Load('C:\Temp\test.xml')
$xmldoc.data.employee
```



Parsing XML File in PowerShell

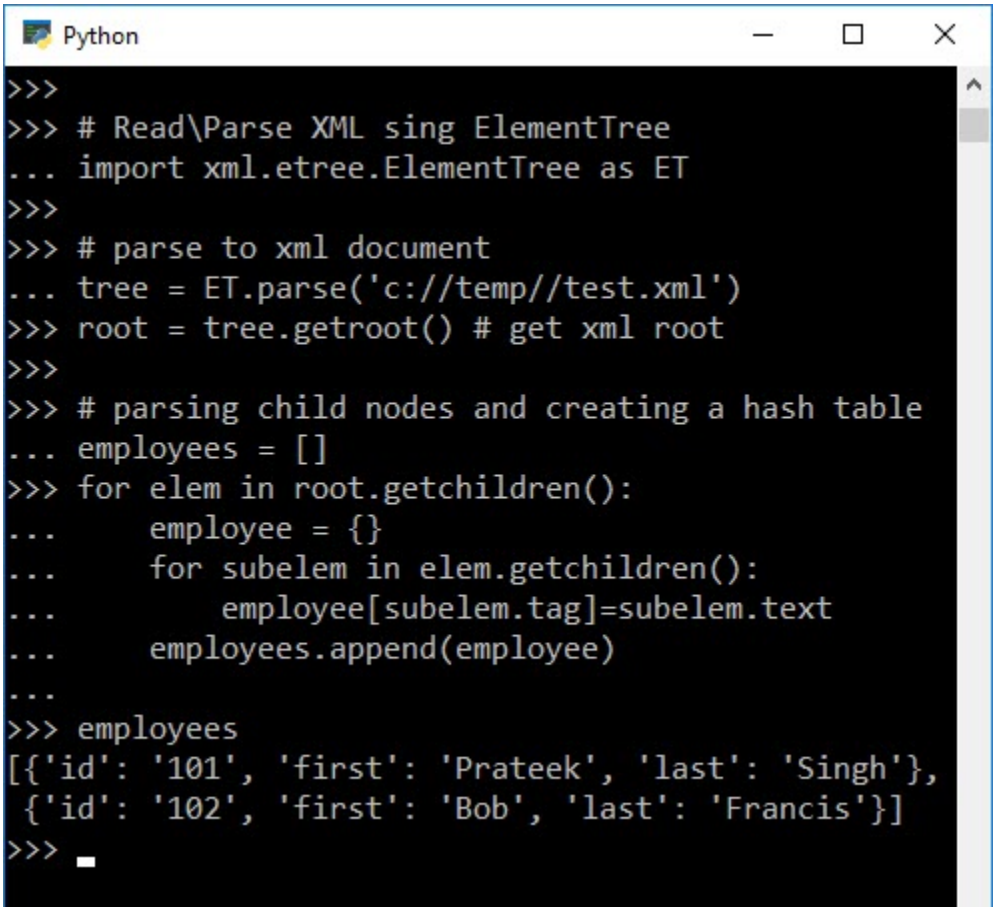
Alternatively, you can also get the content of the XML file using `Get-Content` cmdlet and then cast it to `[XML]` PowerShell type accelerator which is a short hand for class `System.Xml.XmlDocument`.

```
$data = [XML] (Get-Content C:\Temp\test.xml)
$data.data # root element
$data.data.employee
```



Python also supports XML parsing using a inbuilt module called `xml` which has a submodule named: `xml.etree.ElementTree`. You can simply import the `ElementTree` module as a variable, let's suppose 'et' and then use the `parse()` function to load the XML file by providing a valid file path, so that the content of file is parsed into an XML tree. The entry point of this XML tree is root node or the top level node, which can be accessed using the `getroot()` function and to further navigate the child nodes, employ the `getchildren()` function which will return the children of each target parent node. Like in the following example we parsed the XML document and converted it to a list of hash tables holding employee data:

```
# Read\Parse XML using ElementTree
import xml.etree.ElementTree as ET

# parse to xml document
tree = ET.parse('c://temp//test.xml')
root = tree.getroot() # get xml root

# parsing child nodes and creating a hash table
employees = []
for elem in root.getchildren():
    employee = {}
    for subelem in elem.getchildren():
        employee[subelem.tag]=subelem.text
    employees.append(employee)
```

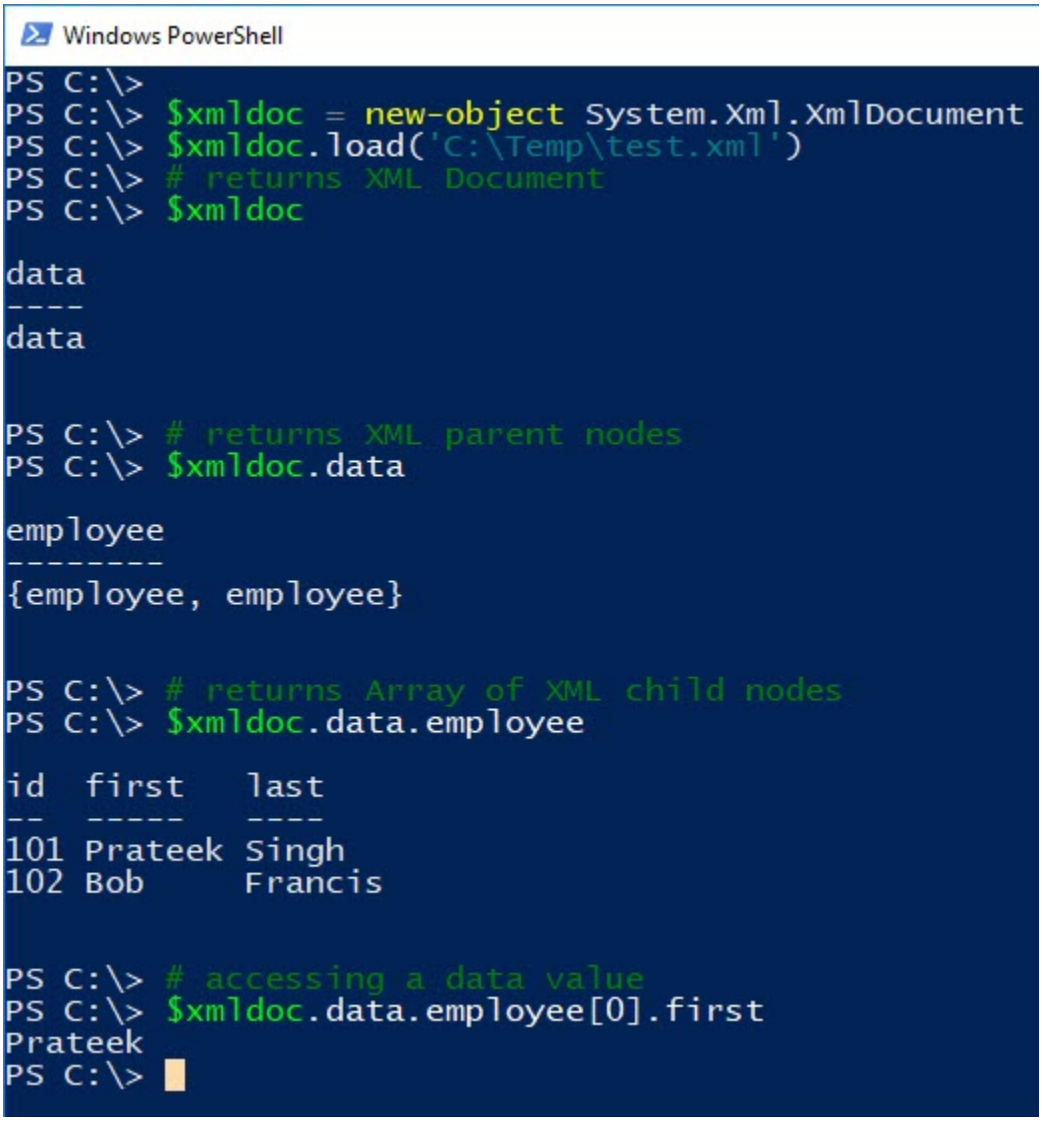


Parsing XML File in Python

Accessing XML as Objects

Once the XML document is parsed in PowerShell like we did in previous examples of this chapter, now it is converted into objects which can be accessed using the Dot (.) operator and can navigate nodes of the parsed XML tree like in the following example:

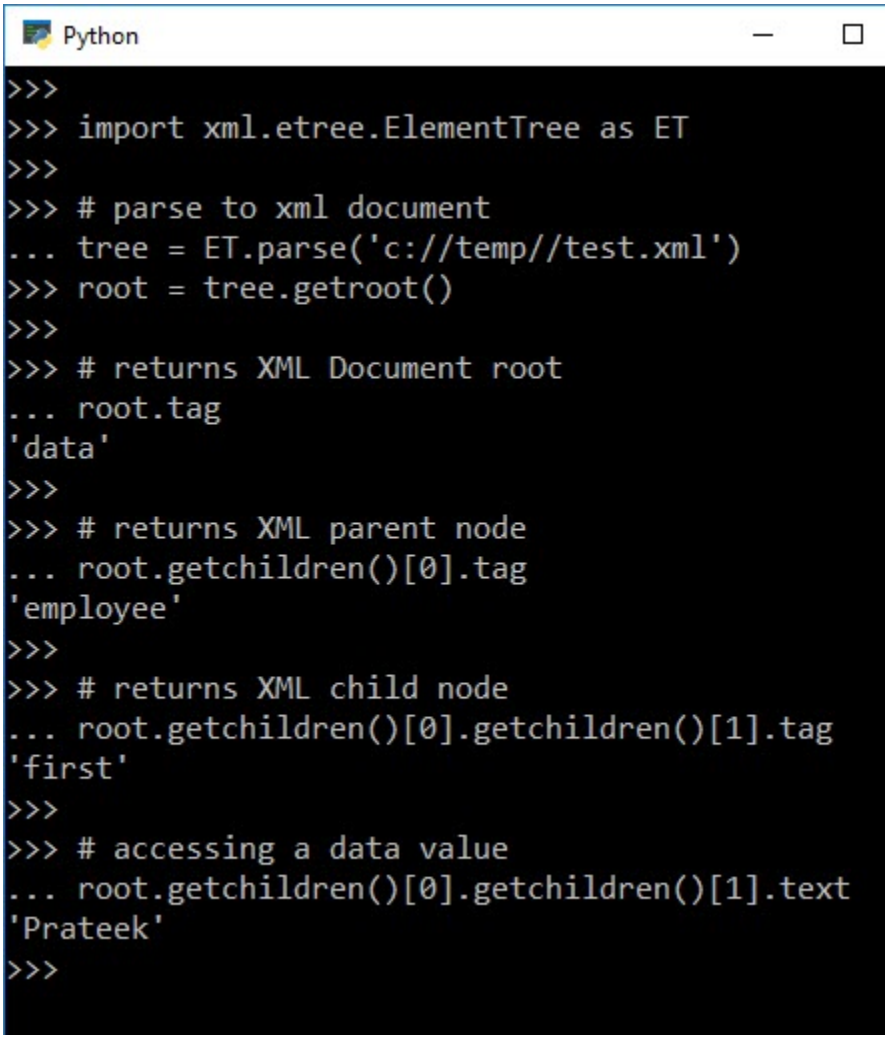
```
$xmldoc = new-object System.Xml.XmlDocument
$xmldoc.load('C:\Temp\test.xml')
# returns XML Document
$xmldoc
# returns XML parent nodes
$xmldoc.data
# returns Array of XML child nodes
$xmldoc.data.employee
# accessing a data value
$xmldoc.data.employee[0].first
```



Accessing parsed XML as objects

Similarly, an XML document parsed and converted to objects using the `ElementTree` module in Python can be accessed and navigated using the Dot operator to get the value between any XML nodes, like in the following example:

```
import xml.etree.ElementTree as ET

# parse to xml document
tree = ET.parse('c://temp//test.xml')
root = tree.getroot()

# returns XML Document root
root.tag

# returns XML parent node
root.getchildren()[0].tag

# returns XML child node
root.getchildren()[0].getchildren()[1].tag

# accessing a data value
root.getchildren()[0].getchildren()[1].text
```

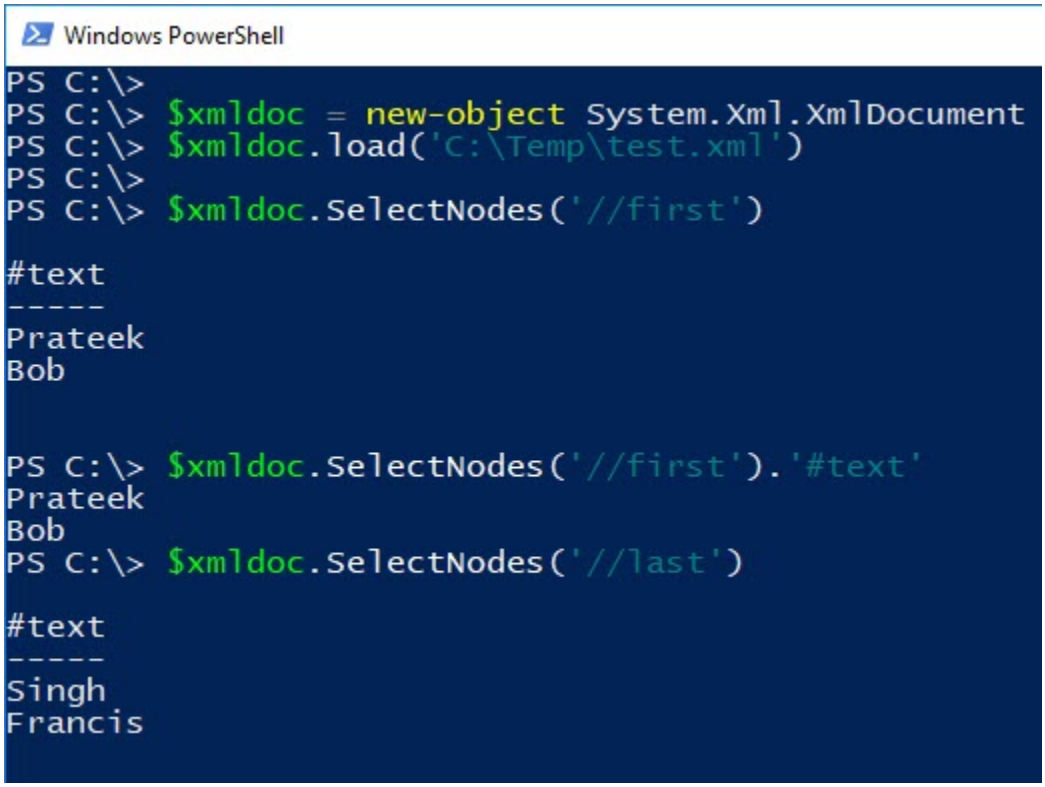


Accessing XML with XPath

One of the most important things of parsing and accessing an XML file is finding XML nodes and data sitting in the respective XML tags. In PowerShell you can use the `SelectNodes()` method by passing an `XPATH` string to return target XML nodes, `XPath`[7] is a query language for selecting nodes from an XML document.

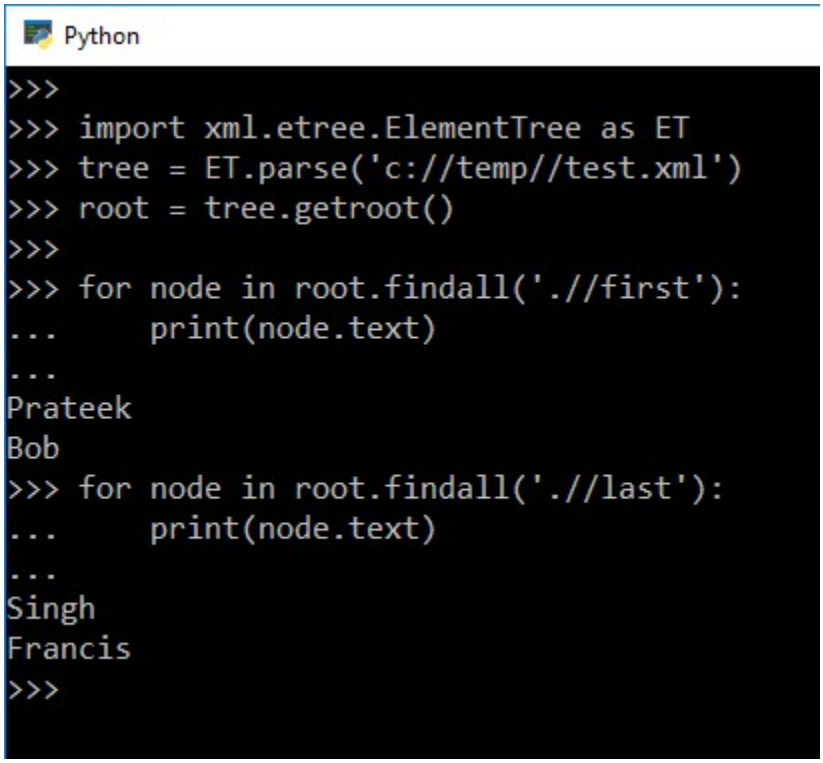
[7] https://en.wikipedia.org/wiki/XPath

```
$xmldoc = new-object System.Xml.XmlDocument
$xmldoc.load('C:\Temp\test.xml')

$xmldoc.SelectNodes('//first')
$xmldoc.SelectNodes('//first').'#text'
$xmldoc.SelectNodes('//last').'#text'
```



Whereas, `root` of Python `xml.etree.ElementTree` object has a `findall()` method which accepts an `XPATH` string as we saw in previous PowerShell example to return targeted XML nodes or elements.

```
import xml.etree.ElementTree as ET
tree = ET.parse('c://temp//test.xml')
root = tree.getroot()

for node in root.findall('.//first'):
    print(node.text)

for node in root.findall('.//last'):
    print(node.text)
```

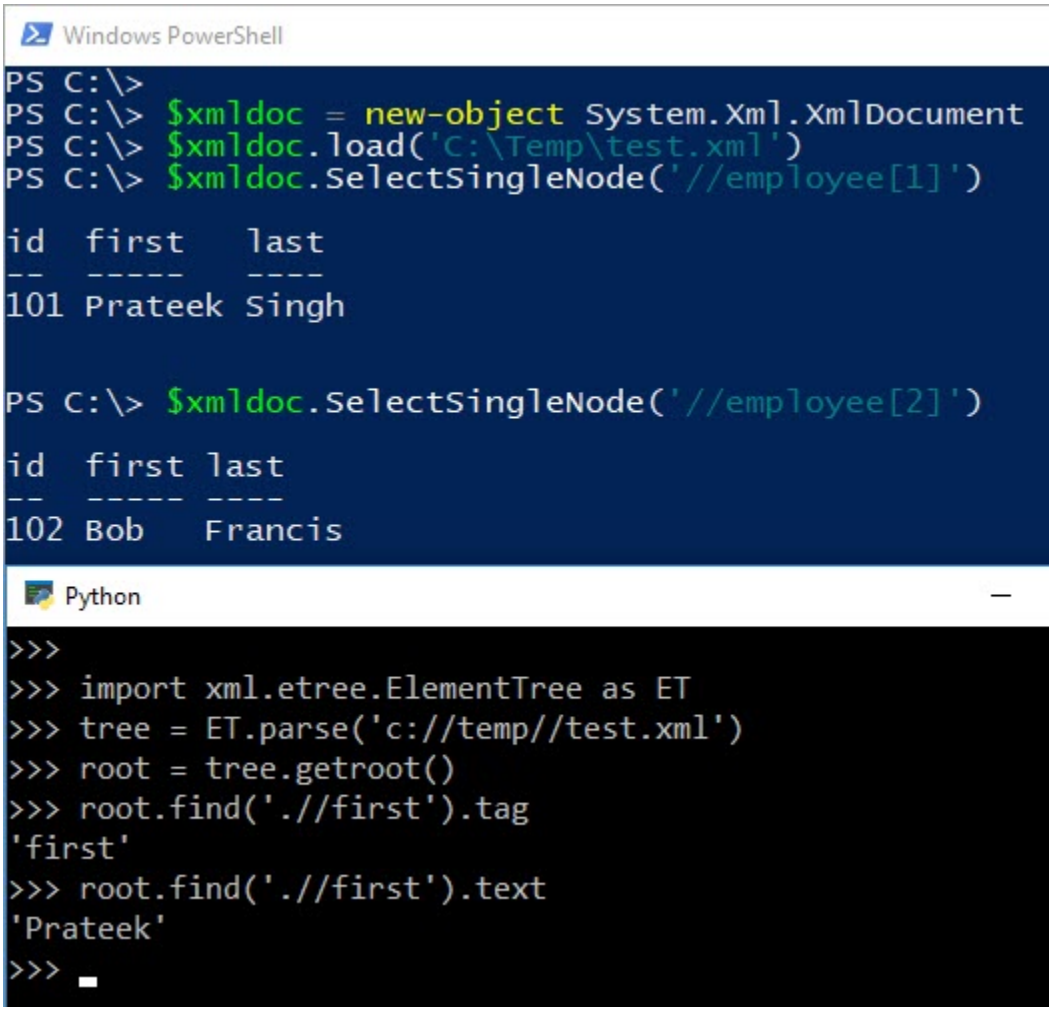


PowerShell also allows you to select unique nodes using the method `SelectSingleNode()` with an `XPATH` string to target a single node. In case there are more than one nodes you can use the brackets `[ ]` and numbers between them just like you access elements of an array, the only difference is that it doesn't start with `'0'`, but `'1'` instead.

```
$xmldoc = new-object System.Xml.XmlDocument
$xmldoc.load('C:\Temp\test.xml')
$xmldoc.SelectSingleNode('//employee[1]')
$xmldoc.SelectSingleNode('//employee[2]')
```

Python in a similar fashion has `find()` method to target single nodes, of a parsed XML tree.

```
import xml.etree.ElementTree as ET
tree = ET.parse('c://temp//test.xml')
root = tree.getroot()
root.find('.//first').tag
root.find('.//first').text
```



Using XML for Object Serialization

Handling XML data is fairly easy in PowerShell compared to any other scripting languages, and PowerShell provides `Export-Clixml` cmdlet to serialize an object and store it in an XML file. Later you can use `Import-Clixml` cmdlet import the XML back to PowerShell in form of objects without losing data integrity once the data is restored.

In terms of computer science translation of data structure or objects into a format that can be stored such as XML or JSON is called Serialization. So when PowerShell objects are translated to XML format using the `Export-Clixml` cmdlet it can be said that PowerShell objects are serialized to an XML file. Like in the following example we captured few Service objects in PowerShell and serialized them to an XML file: `C:\Temp\services.xml` using the `Export-Clixml` cmdlet.

```
Get-Service *update* | Select-Object name, status -First 2
Get-Service *update* | Select-Object name, status -First 2 | Export-Clixml C:\Temp\s\
ervices.xml
```

PowerShell takes care of all the serialization and the XML file would look like this:

```
<Objs Version="1.1.0.1" xmlns="http://schemas.microsoft.com/powershell/2004/04">
  <Obj RefId="0">
    <TN RefId="0">
      <T>Selected.System.ServiceProcess.ServiceController</T>
      <T>System.Management.Automation.PSCustomObject</T>
      <T>System.Object</T>
    </TN>
    <MS>
      <S N="Name">AdobeFlashPlayerUpdateSvc</S>
      <Obj N="Status" RefId="1">
        <TN RefId="1">
          <T>System.ServiceProcess.ServiceControllerStatus</T>
          <T>System.Enum</T>
          <T>System.ValueType</T>
          <T>System.Object</T>
        </TN>
        <ToString>Stopped</ToString>
        <I32>1</I32>
      </Obj>
    </MS>
  </Obj>
  <Obj RefId="2">
    <TNRef RefId="0" />
    <MS>
      <S N="Name">AdobeUpdateService</S>
      <Obj N="Status" RefId="3">
        <TNRef RefId="1" />
        <ToString>Running</ToString>
        <I32>4</I32>
      </Obj>
    </MS>
  </Obj>
</Objs>
```

On other hand restoring the serialized XML data to PowerShell objects is called de-serialization, and can be achieved using the `Import-Clixml` cmdlet by providing the full file path to return PowerShell objects we captured from services, like in the following example:

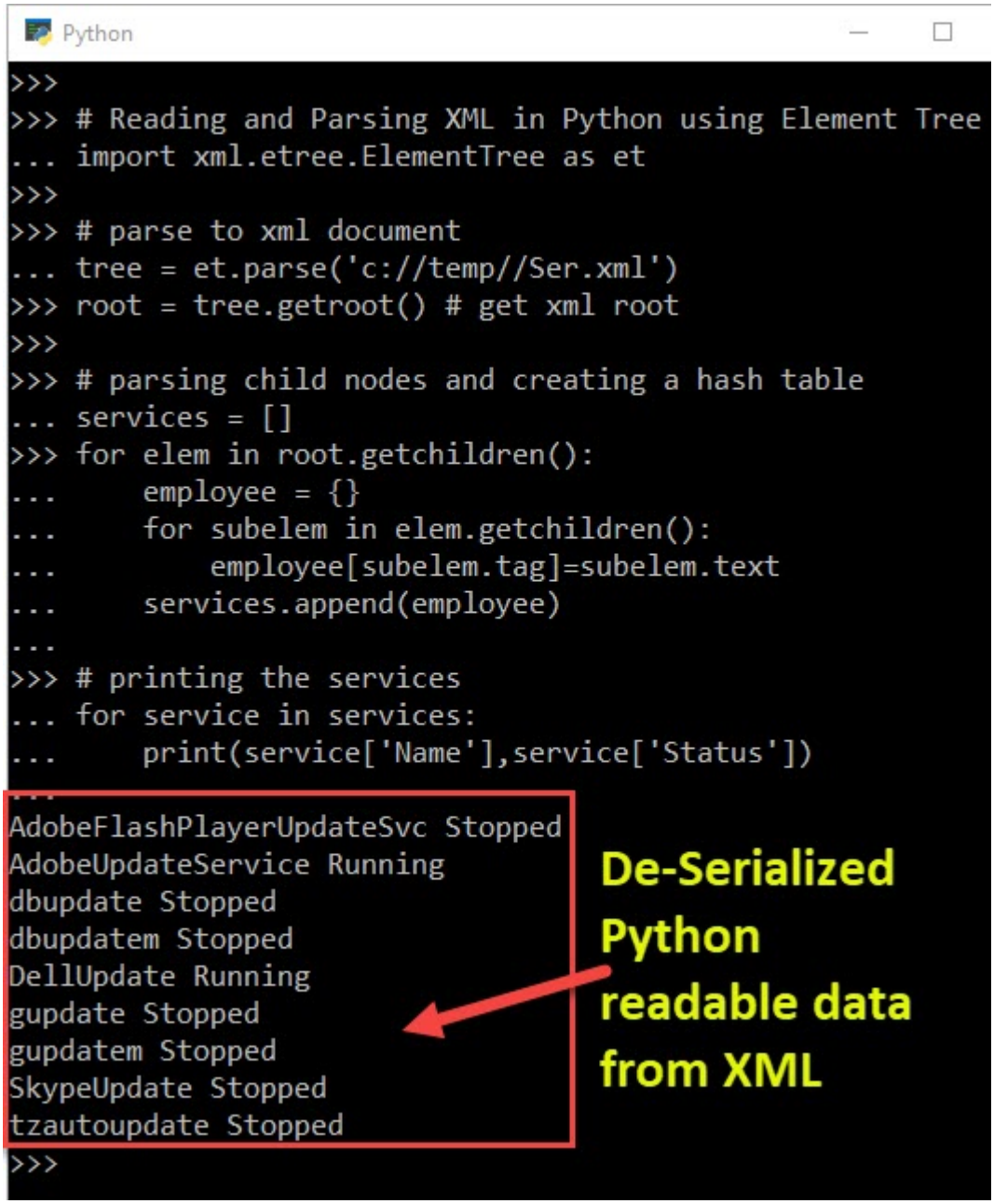
```
PS C:\> Import-Clixml C:\Temp\services.xml

Name                        Status
----                        ------
AdobeFlashPlayerUpdateSvc Stopped
AdobeUpdateService        Running
```

Let's take the same example in Python and capture some services using a third party module called `wmi`, and this module can be installed on your machine using `pip` installer: `python.exe -m pip install wmi`. Once we have the services, we will use the `Element()` and `SubElement()` functions of `xml.etree.ElementTree` module to create an XML tree, in words we are building an XML tree. After we have defined the complete XML tree, then we will use the `write()` function to serialize this tree object into XML format and write it to a file.

```
# list services
import wmi
import xml.etree.ElementTree as et

conn = wmi.WMI()
root = et.Element("Services") # root

# creating XML tree
for s in conn.Win32_Service():
    if 'Update' in s.Name or 'update' in s.Name:
        elem = et.Element("Service")
        root.append(elem)
        name = et.SubElement(elem, "Name")
        name.text = s.Name
        status = et.SubElement(elem, "Status")
        status.text = s.State

# finishing the tree hierarchy
tree = et.ElementTree(root)

# writing the tree to XML file: Serialization
tree.write('C://Temp//Ser.xml')
```

Now to de-serialize the XML content in python usable format, you have to parse it as we did in the first sub-section of this chapter and you would be good to go like in the following example:

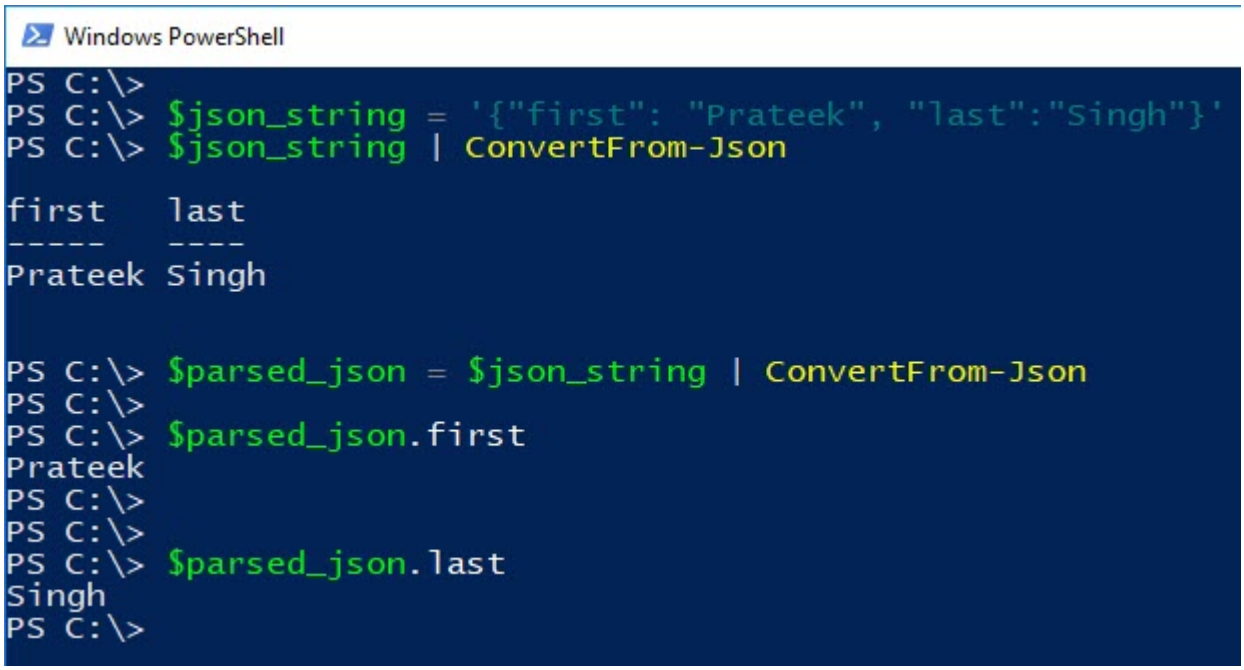
```
# Reading and Parsing XML in Python using Element Tree
import xml.etree.ElementTree as et

# parse to xml document
tree = et.parse('c://temp//Ser.xml')
root = tree.getroot() # get xml root

# parsing child nodes and creating a hash table
services = []
for elem in root.getchildren():
    employee = {}
    for subelem in elem.getchildren():
        employee[subelem.tag]=subelem.text
    services.append(employee)

# printing the services
for service in services:
    print(service['Name'],service['Status'])
```



Parsing JSON

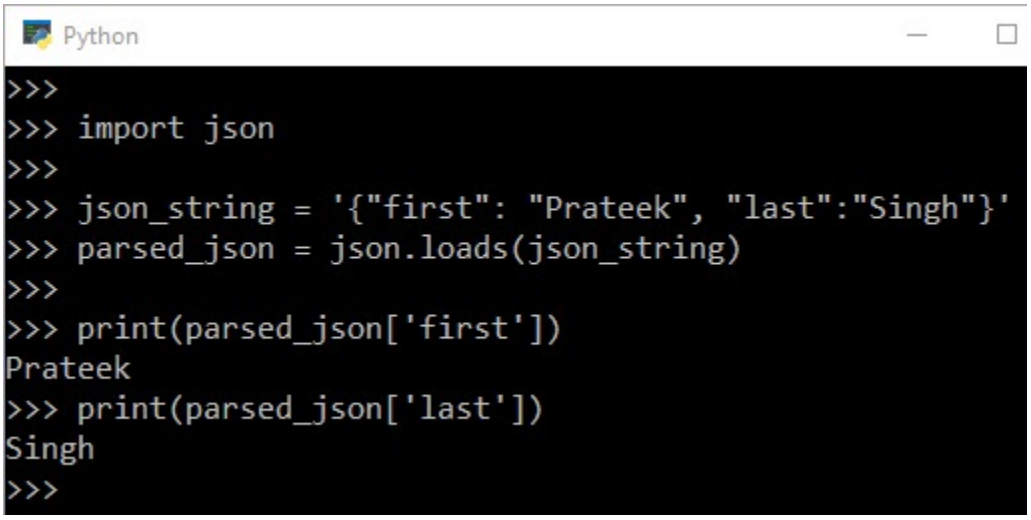
Parsing a data in a JSON format in PowerShell is as easy as piping the JSON string to `ConvertTo-JSON` cmdlet which will automatically parse the JSON content and return the PowerShell objects, like in the following example:

```powershell
$json_string = '{"first": "Prateek", "last":"Singh"}'

# parses the JSON string to objects
$parsed_json = $json_string | ConvertFrom-Json

# accessing the parsed JSON data
$parsed_json.first
```



Python on other hand has an inbuilt module known as `json` to parse data in JSON formats and files. Simply import this module and use the `loads()` function which will return Python objects after parsing the JSON data.

```python
import json

json_string = '{"first": "Prateek", "last":"Singh"}'

# loads and parses the JSON
parsed_json = json.loads(json_string)

# accessing the parsed JSON like a hash table
print(parsed_json['first'])
```



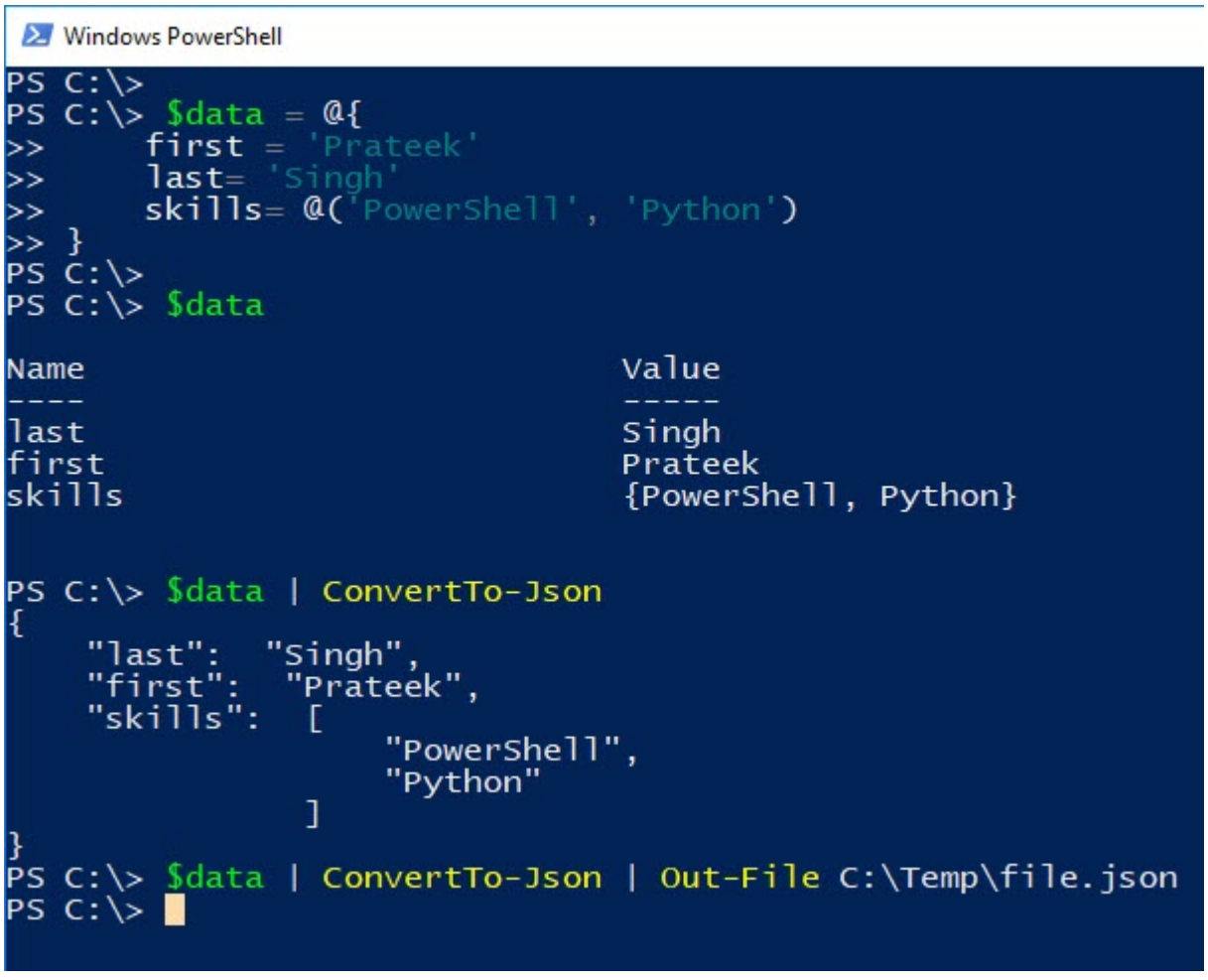
JSON Serialization and De-Serialization

In order to store and persist data in JSON format, both PowerShell and Python supports serialization which can convert objects into JSON formatted data files. PowerShell has a `ConvertTo-JSON` cmdlet which can convert any object to a JSON equivalent, which can then be piped to a file to be written using the `Out-File` cmdlet like in the following example we serialized PowerShell objects to JSON formatted file:

```
$data = @{
    first = 'Prateek'
    last= 'Singh'
    skills= @('PowerShell', 'Python')
}

# writing the objects to a file serialized in XML format
$data | ConvertTo-JSON | Out-File C:\Temp\file.json
```

Now if you open the file `C:\Temp\file.json` you will JSON data like this:

```
{
    "last":  "Singh",
    "first":  "Prateek",
    "skills":  [
                    "PowerShell",
                    "Python"
                ]
}
```

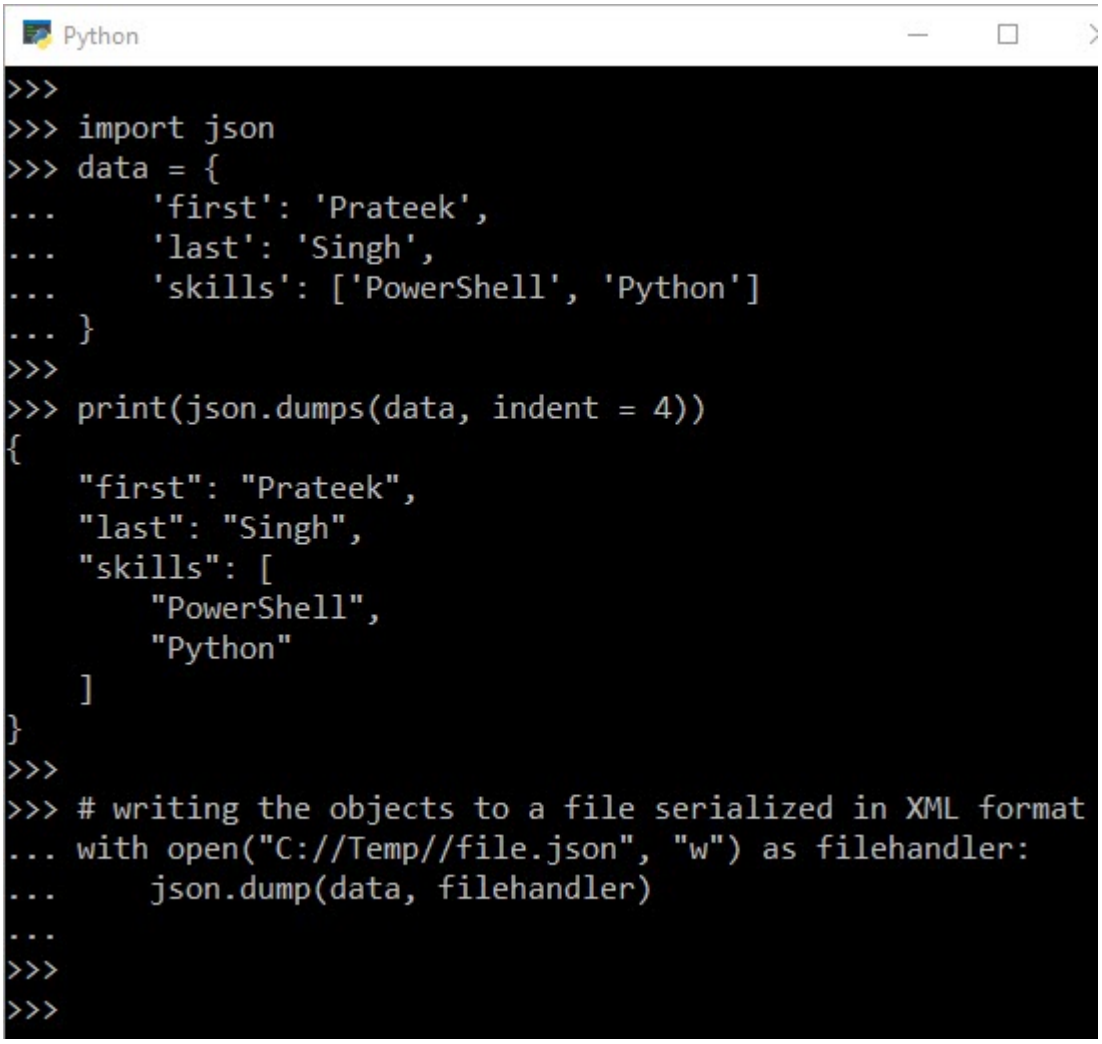


Whereas, in Python we employ the `dump()` function of `json` module to serialize Python objects to JSON formatted data and then we write it to a file.

```
import json
data = {
    'first': 'Prateek',
    'last': 'Singh',
    'skills': ['PowerShell', 'Python']
}

print(json.dumps(data, indent = 4))

# writing the objects to a file serialized in XML format
with open("C://Temp//file.json", "w") as filehandler:
    json.dump(data, filehandler)
```

The serialized JSON format looks something like this:

```
{"first": "Prateek", "last": "Singh", "skills": ["PowerShell", "Python"]}
```



In order to use this serialized JSON data in your PowerShell program, you have to restore them in their original form of PowerShell objects, this is also known as De-serialization. PowerShell can simply convert the content any such JSON file into PowerShell objects using the `ConvertFrom-Json` cmdlet like in the following example we are capturing the content of JSON file using the `Get-Content` cmdlet and then piping it to `ConvertFrom-JSON` cmdlet, so that it can be de-serialized or in other word converted from JSON format to PowerShell Objects:

```
1  Get-Content C:\Temp\file.json | ConvertFrom-Json
```

```
Windows PowerShell
PS C:\>
PS C:\> Get-Content C:\Temp\file.json | ConvertFrom-Json

last  first   skills
----  -----   ------
Singh Prateek {PowerShell, Python}


PS C:\> $data = Get-Content C:\Temp\file.json | ConvertFrom-Json
PS C:\>
PS C:\> $data.first
Prateek
PS C:\> $data.skills
PowerShell
Python
PS C:\>
```

Python's `json` module has a `load()` function, which has similar de-serializing capabilities and can translate JSON data to Python objects as demonstrated in following code sample. All you have to do is open the file in read-only mode and pass the file handler object to the `load()` function, which will then convert the JSON formatted content of the file into Python usable data structure.

```
1  import json
2  with open("C://Temp//file.json", "r") as filehandler:
3      data = json.load(filehandler)
4
5  data['first']
6  data['skills']
```

```
Python
>>>
>>> with open("C://Temp//file.json", "r") as filehandler:
...     data = json.load(filehandler)
...
>>> data['first']
'Prateek'
>>> data['skills']
['PowerShell', 'Python']
>>>
>>>
```

Chapter 18 - Reading and Writing CSV Files

Exchange of information into and out of programs is the requirement for most of programming and scripting languages of the world. The most common approach of this information exchange is through text files and one of the most popular formats for exchanging such data is the CSV format. So in this chapter, we are going to understand how to parse, write and append CSV files using PowerShell and Python.

What is CSV

`CSV` is short for '`comma-separated values`', which is a tabular data that has been saved as plaintext data separated by commas (`','`) or any other delimiter. For example, we have data in the following table which has header and data elements, which can be converted into CSV data where each row of a table is a newline and has data separated with a comma.

Tabular Data:

Column1	Column2	Column3
Data1	Data2	Data3
Data1	Data2	Data3

CSV Data:

```
Column1,Column2,Column3
Data1,Data2,Data3
Data1,Data2,Data3
```

Reading CSV File

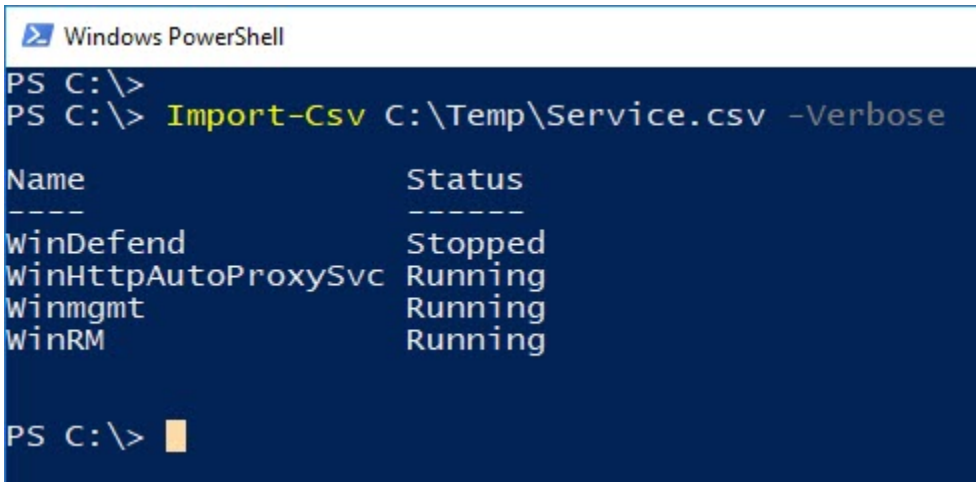
Using PowerShell cmdlet `Import-CSV`

Reading CSV data from a file is fairly easy in PowerShell and all you need to do is use the `Import-CSV` cmdlet with a valid file path. PowerShell will read the CSV file and convert it to PowerShell Objects and return objects on your console, let's suppose we have following CSV data saved as filename: `Service.csv` .

```
"Name","Status"
"WinDefend","Stopped"
"WinHttpAutoProxySvc","Running"
"Winmgmt","Running"
"WinRM","Running"
```

To read or parse the CSV formatted files into PowerShell objects, run `Import-CSV <File Name>`:

```
Import-Csv C:\Temp\Service.csv
```



Using Python's `csv` Library

Python has an inbuilt `csv` library to work with CSV files that can both read and write data from/to the CSV files. First, we import the `csv` module and then open the CSV file in `read-only` mode so that the file can be read and after that, we pass the file handler object to the `reader()` function of `csv` library which will parse the CSV format. Then we iterate through each row of parsed CSV file as demonstrated in the following example.

```
import csv

# open the file in read-only mode
with open('c://Temp//Service.csv','r') as f:
    # read the content of the csv file
    reader = csv.reader(f, delimiter=',')
    for row in reader:
        # return each row as a list
        print(row)
```

Please note that the named parameter: `delimiter` in the `reader()` function is totally optional because default separator character or delimiter is the comma (`','`), but you can also use other popular delimiters, which include the tab (`\t`), colon (`:`) and semi-colon (`:`) characters.

Python

```
>>>
>>> import csv
>>>
>>> # open the file in read-only mode
... with open('c://Temp//Service.csv','r') as f:
...     # read the content of the csv file
...     reader = csv.reader(f, delimiter=',')
...     for row in reader:
...         # return each row as a list
...         print(row)
...
['Name', 'Status']
['WinDefend', 'Stopped']
['WinHttpAutoProxySvc', 'Running']
['Winmgmt', 'Running']
['WinRM', 'Running']
>>>
```

In the above row each row is a list of items per column, that means you can even access each column in a row as an index of a list, like the following example `column-1` and `column-2` are accessed and processed using `row[0]` and `row[1]` respectively:

```
1  import csv
2
3  # open the file in read-only mode
4  with open('c://Temp//Service.csv','r') as f:
5      # read the content of the csv file
6      reader = csv.reader(f, delimiter=',')
7      for row in reader:
8          print('Service:{0} is {1}'.format(row[0],row[1]))
```

Another approach can be reading the CSV data directly into a dictionary using the `DictReader` class of `csv` library, instead of processing a list of individual String elements on each row of the CSV file. Once the content of CSV file is converted to Python dictionary, then access the individual elements of a row by passing name of the column as key to the dictionary as demonstrated in the following example:

```
import csv

# open the file in read-only mode
with open('c://Temp//Service.csv','r') as f:
    # read the content of the csv file
    reader = csv.DictReader(f, delimiter=',')
    for row in reader:
        print('Service:{0} is {1}'.format(row['Name'],row['Status']))
```

```
Python
>>>
>>> # open the file in read-only mode
... with open('c://Temp//Service.csv','r') as f:
...     # read the content of the csv file
...     reader = csv.DictReader(f, delimiter=',')
...     for row in reader:
...         print('Service:{0} is {1}'.format(row['Name'],row['Status']))
...
Service:WinDefend is Stopped
Service:WinHttpAutoProxySvc is Running
Service:Winmgmt is Running
Service:WinRM is Running
>>>
```

Writing CSV File

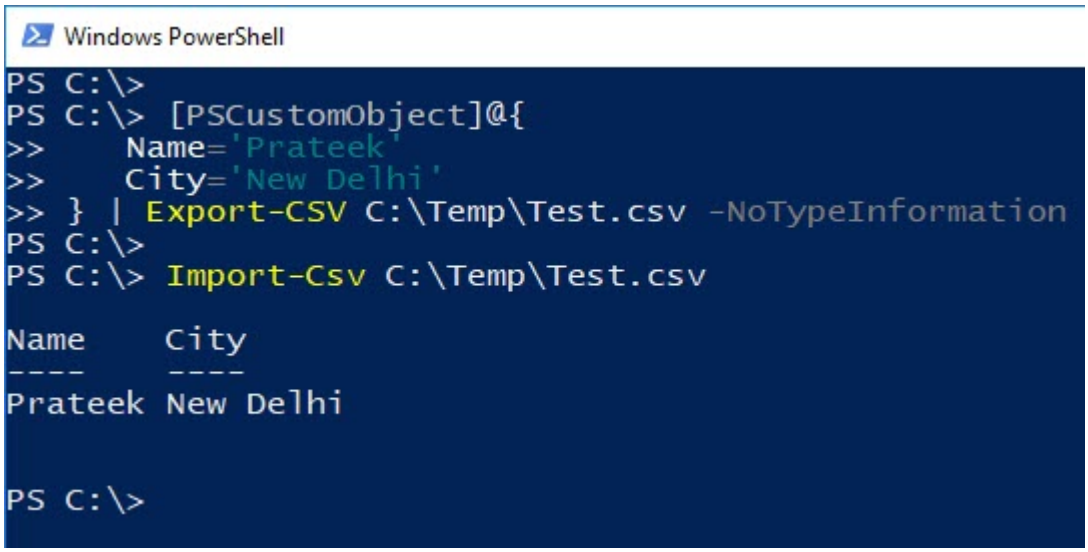
Using PowerShell cmdlet `Export-CSV`

PowerShell Objects can be simply piped to the `Export-CSV` cmdlet with a valid file name, and it will convert the object into CSV format and write to the file. Like in the following example first we capture some services running on a machine, with specific properties and then export it to a file through the pipeline using the `Export-CSV` cmdlet. The use of `-NoTypeInformation` switch parameter indicates that this cmdlet omits the type information from the CSV file, which is `#TYPE` followed by the fully-qualified name of the type of the object in the first line of the CSV file.

```
Get-Service win* | Select-Object Name, Status | Export-Csv C:\Temp\Service.csv -NoTy\
peInformation
```

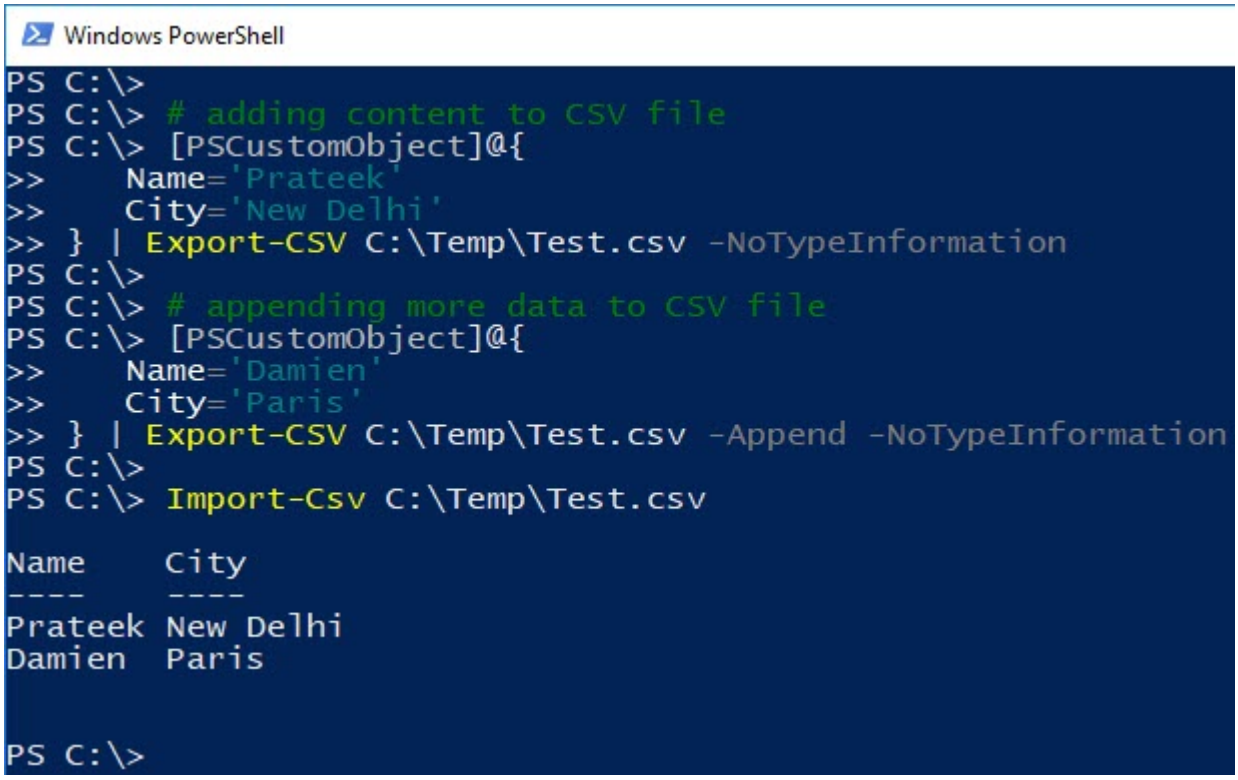
You can even convert dictionaries to PowerShell objects by casting it to `[PSCustomObject]` type accelerator, which will convert it into an object and then you can export it to a csv file using `Export-CSV` cmdlet.

```
[PSCustomObject]@{
    Name='Prateek'
    City='New Delhi'
} | Export-CSV C:\Temp\Test.csv -NoTypeInformation
```



PowerShell also supports appending the content to a CSV file, all you need to do is add the `-Append` switch while writing the content using `Export-CSV` cmdlet and the new data would be written at the end of the CSV file. Without the `-Append` switch content would be overwritten to the file, and any previous data would be lost.

```
# adding content to CSV file
[PSCustomObject]@{
   Name='Prateek'
   City='New Delhi'
} | Export-CSV C:\Temp\Test.csv -NoTypeInformation


# appending more data to end of the CSV file
[PSCustomObject]@{
   Name='Damien'
   City='Paris'
} | Export-CSV C:\Temp\Test.csv -Append -NoTypeInformation

# read the CSV file
Import-Csv C:\Temp\Test.csv
```



Using Python's `csv` Library

Python's `csv` library has a `writer` function that can be used to write CSV data to a file. To write a single row we use `writerow()` function by passing a list of header values, but in case we want to pass a list of rows at once instead of one row at a time, we can also utilize another function:`writerows()` as demonstrated in the following example:

```
1  import csv
2
3  header = ['Name','Branch']
4  rows = [ ['Prateek','IT'],
5           ['Damien','HR'],
6           ['John', 'COE'] ]
7
8  with open('c://Temp//Employee.csv', 'w', newline='') as f:
9      writer = csv.writer(f, delimiter=',')
10     writer.writerow(header)
11     writer.writerows(rows)
```

Please note that in above example, when we are opening the file to write the CSV data, then I've also specified a named parameter: `newline=''` with an empty string, this is just to make sure that a

newline character is not added after each row. If you omit this parameter then you may see proper CSV data in a text editor, but a new line would be added if you open the same file in Microsoft Excel spreadsheet. Ideally, the CSV file: `Employee.csv` generated in the above example should look like this:

```
Name,Branch
Prateek,IT
Damien,HR
John,**COE**
```

`csv` library in Python is capable to write dictionaries directly to a CSV formatted file, which is achieved by instantiating the `DictWriter` class by passing the headers to `fieldnames` parameter. Then use `writeheader()` function to write these headers to the CSV file, after which you can add rows to the CSV file using the `writerow()` function by passing dictionaries to it as demonstrated in the following example.

```
import csv

with open('c://Temp//Employee.csv', mode='w', newline='') as f:
    header = ['Name', 'Branch']
    writer = csv.DictWriter(f, fieldnames=header)
    writer.writeheader()
    writer.writerow({'Name': 'Prateek', 'Branch': 'IT'})
    writer.writerow({'Name': 'Damien', 'Branch': 'HR'})
    writer.writerow({'Name': 'John', 'Branch': 'COE'})
```

In case you want to append CSV content to a file instead of overwriting it, then please make sure to open the file in `append` mode: `mode='a'` not in write mode.

```
import csv

with open('c://Temp//Employee.csv', mode='a', newline='') as f:
    header = ['Name', 'Branch']
    writer = csv.DictWriter(f, fieldnames=header)
    writer.writerow({'Name': 'Prateek', 'Branch': 'IT'})
    writer.writerow({'Name': 'Damien', 'Branch': 'HR'})
    writer.writerow({'Name': 'John', 'Branch': 'COE'})
```

Chapter 19 - Error Handling

Handling errors or exceptional conditions is one of the key parts of any programming language. Even small scripts and programs should be ideally written keeping in mind that they should be able to catch any exception and handle them gracefully in an expected manner. So in this chapter, we are going to delve deeper into types of errors and handling them, we will learn to throw, catch and create custom errors.

Types of Errors

Generally, there are 3 types of errors: Syntax, Runtime and Logical errors. Let's look into each of these errors one at a time.

Syntax errors

In terms of any language (like English): Arrangement of words and phrases to create well-formed sentences that can be understood is called Syntax, but in terms of Computer Science: Syntax is a structure of statements. Two important take away words from these two definitions are: 'Well-formed' and 'Structure'.

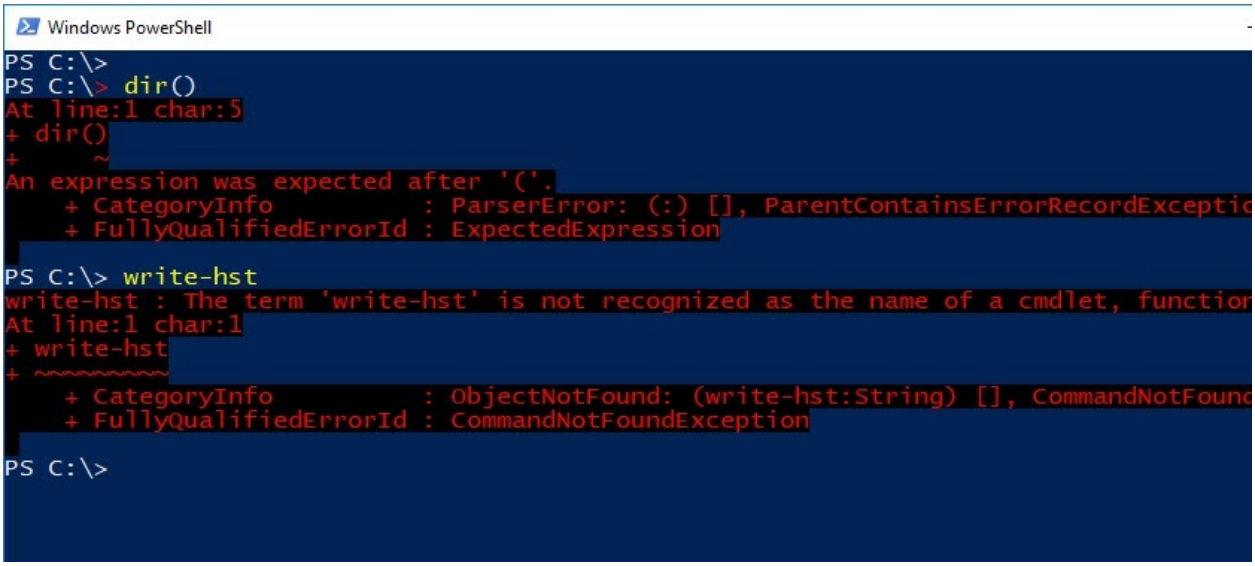
So in any programming language, a Syntax error is usually observed when the parser detects an incorrect statement or finds programming construct such as a parenthesis or a bracket missing in the program. In other words, a syntax error is a grammatical error for a programming language which expects developers to provide statements in specific structure with all proper programming constructs and if this is not the case an error will be thrown. Following are some common examples of Syntax errors:

- Misspelled cmdlets, variable or name of a function.
- Missing semicolons or improper indentation.
- Improperly matched parentheses `( )`, square brackets `[ ]`, and curly braces `{ }`

Let's see some examples to understand this better.

PowerShell:

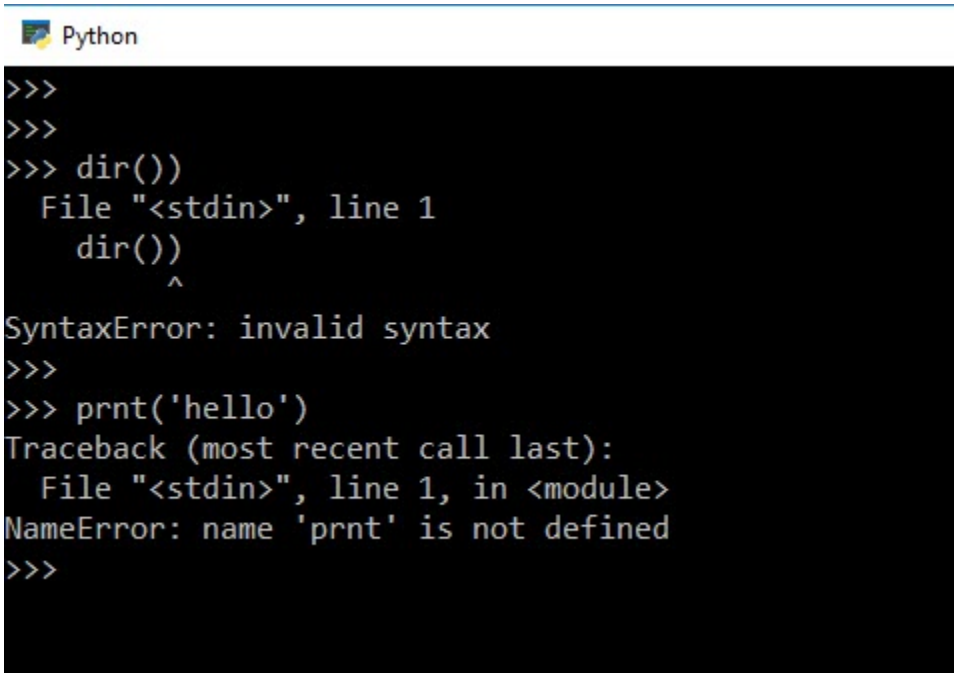
```
dir()
write-hst
```



The first example in the above PowerShell sample is using the `dir` alias of `Get-ChildItem` cmdlet, since the parentheses `()` was not expected after a cmdlet so you will run into Syntax errors. In the Second example, instead of using `Write-Host` cmdlet it was misspelled as `write-hst` which the parser couldn't understand and we ran into an error again.

Python:

```
>>> dir())
  File "<stdin>", line 1
    Get-ChildItem())
         ^
SyntaxError: invalid syntax
>>>
>>> prnt('hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'prnt' is not defined
```



The first example in the above python sample has a an extra closing parentheses `')'` which was not expected there and a `'SyntaxError: invalid syntax'` was thrown. In the second example, we misspelled the `print()` function so the parser couldn't understand it and throws an `'NameError: name 'prnt' is not defined'`.

Runtime errors

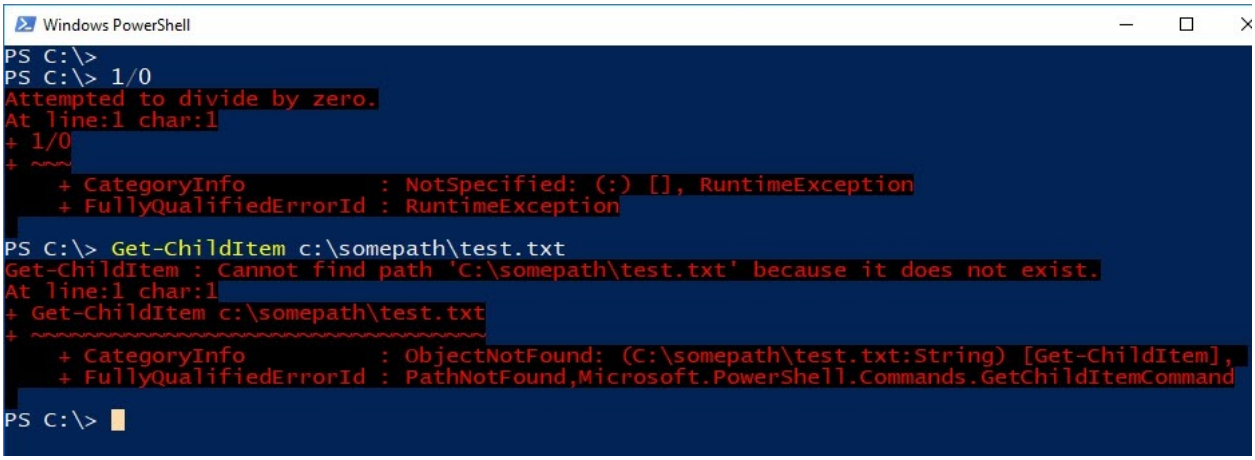
Runtime errors like the name suggests, can only occur while the program is running, assuming that there are no syntax errors. Runtime errors are usually observed when the computer program is asked to perform something that it is unable to handle reliably. Following are some common examples of Runtime errors:

- File doesn't exist
- Divided by zero

Let's see some examples to understand this better:

PowerShell:

```powershell
# dividing a number by zero.
1/0

# accessing a file that doesn't exist
Get-ChildItem c:\somepath\test.txt
```



Python:

```python
# dividing a number by zero.
1/0

# accessing a file that doesn't exist
open('c://somepath//test.txt')
```

Logic errors

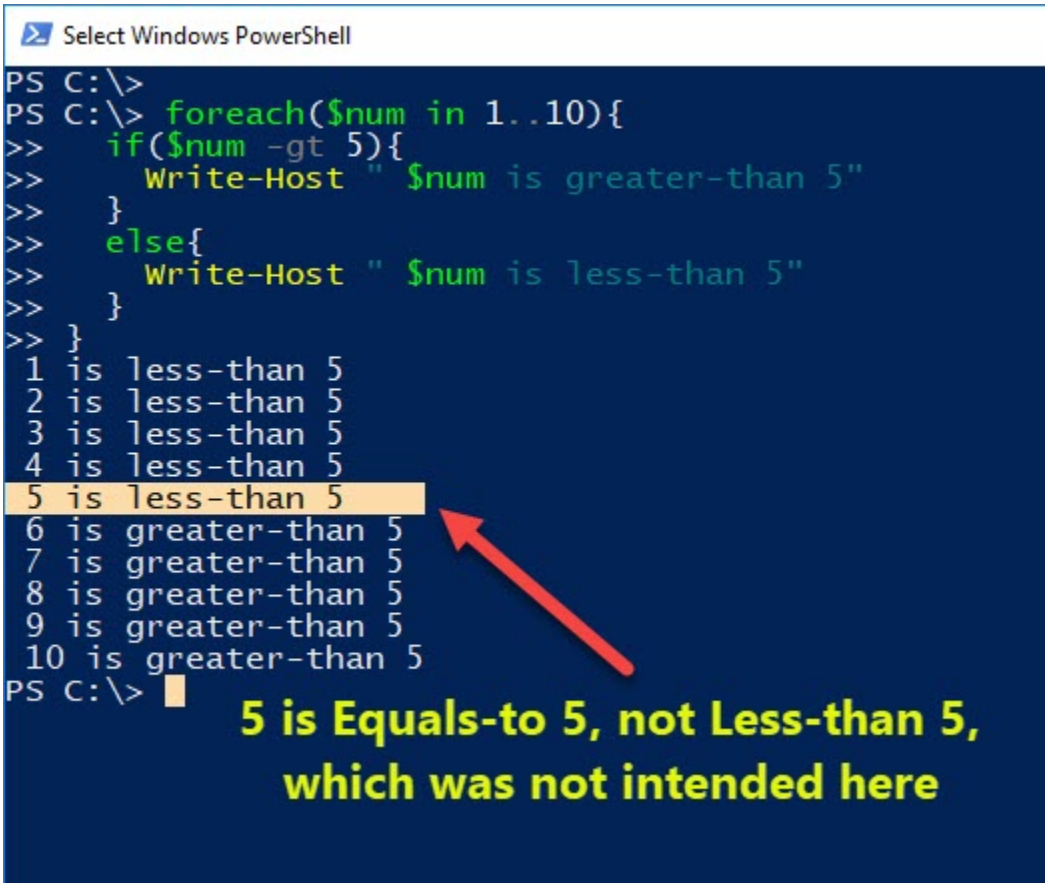
In terms of computer programming, a logical error is a bug in a program that causes it to operate incorrectly or behave abnormally to produce unintended results. A logical error may not be immediately recognized, because unlike syntax or runtime errors a logical error doesn't terminate abnormally and is considered correct as per the programming language. Following are some common examples of logical errors:

- Performing Addition in place of subtraction
- Using assignment operator `'='` in place of equals-to (`'=='` or `'-eq'`)
- Using the wrong variable or file name.

Let's see some examples to understand this better.

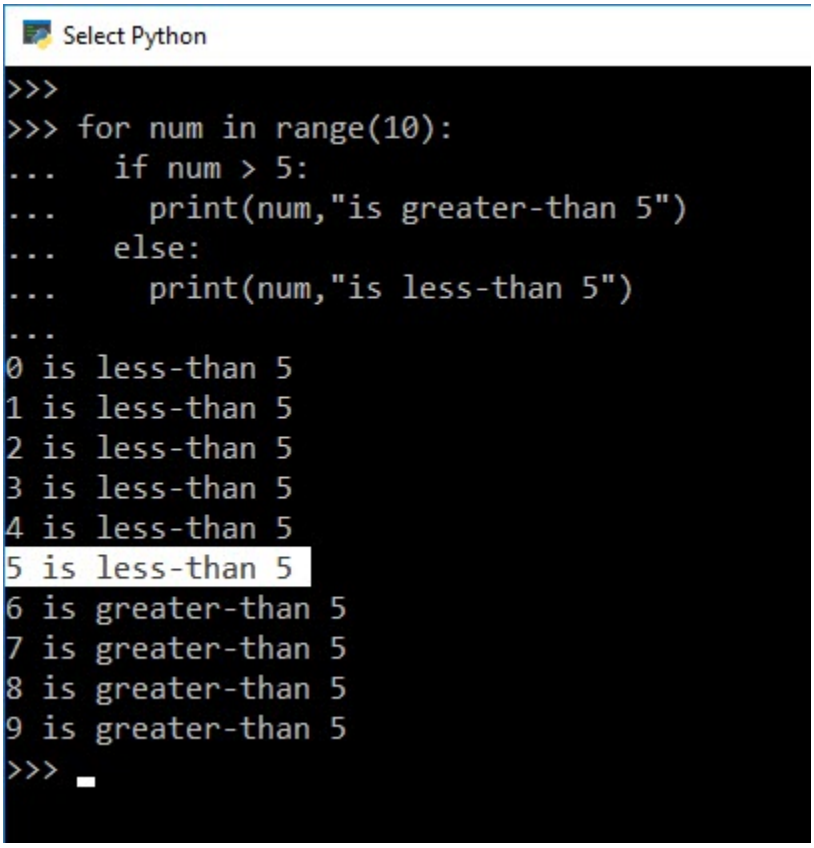
```powershell
foreach($num in 1..10){
  if($num -gt 5){
    Write-Host " $num is greater-than 5"
  }
  else{
    Write-Host " $num is less-than 5"
  }
}
```

The logical error in above example is that the `else` statement should not execute and the condition should evaluate to `$True` when value of variable:`$name` is equals to `5` because `5` is not less than `5`.



A similar implementation of the logical error in Python, which will run just fine without throwing any errors, but the result it generates is unintended and logically incorrect.

```
for num in range(10):
  if num > 5:
    print(num,"is greater-than 5")
  else:
    print(num,"is less-than 5")
```



Raising an Exception or Throwing an error

Sometimes it is a requirement for a developer to explicitly raise or throw an exception so that it can be handled properly. Because otherwise the programming language may or may not throw that error.

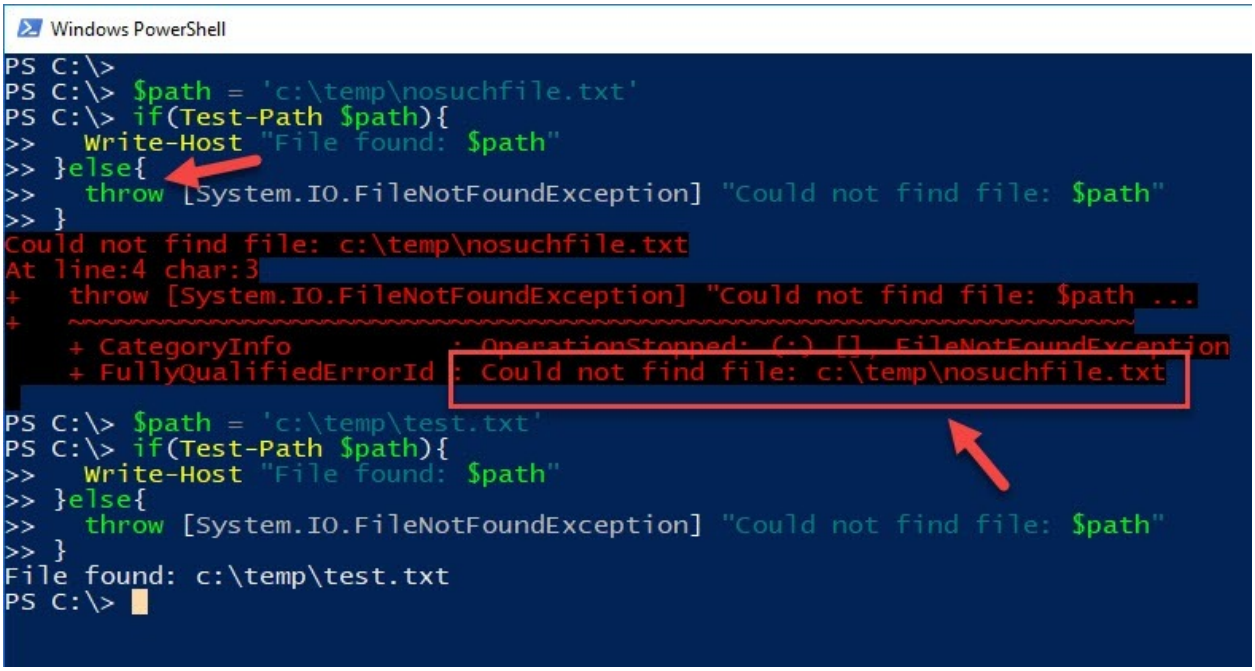
PowerShell has a `throw` keyword, which can be used to create your own exception events.

Syntax:

```
throw [ExceptionClass] "Your Error Message"
```

In the below example, we raised an `FileNotFoundException`, when no such file was found the program.

```
$path = 'c:\temp\nosuchfile.txt'
if(Test-Path $path){
  Write-Host "File found: $path"
}else{
  throw [System.IO.FileNotFoundException] "Could not find file: $path"
}
```



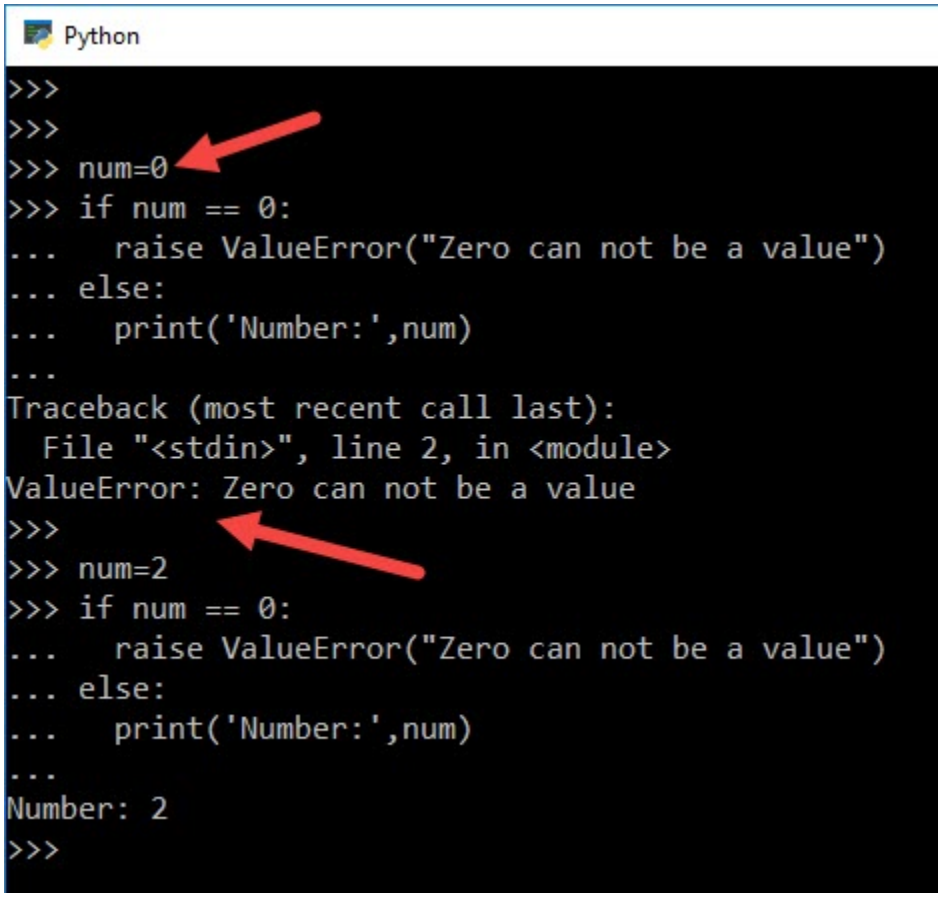
To raise an exception from Python you have to utilize the `raise` keyword.

Syntax:

```
raise ExceptionClass("Your Error Message")
```

Like in the following Python example, we raised a `ValueError` using the `raise` keyword, when the a numerical value did not meet a specific criteria.

```
num=2
if num == 0:
  raise ValueError("Zero can not be a value")
else:
  print('Number:',num)
```



Handling Exceptions using Try..catch..finally

In any programming language, the way exception handling works is that you first attempt executing a section of the program in a `try` block and if it throws an error you can handle it in a `catch` block, and at last you perform some steps final mandatory steps in the `finally` block. Let us look into each of these blocks in detail one at a time.

Try Block

The `'try'` block defines a section of code which you want the programming language to monitor for any errors or exceptions. Following is the PowerShell Syntax for the `try` block:

```
try{
  # statements
}
```

On other hand Python Syntax is also almost similar, except the colon `' : '` after the `try` keyword and the indentation instead of curly brackets `{ }`.

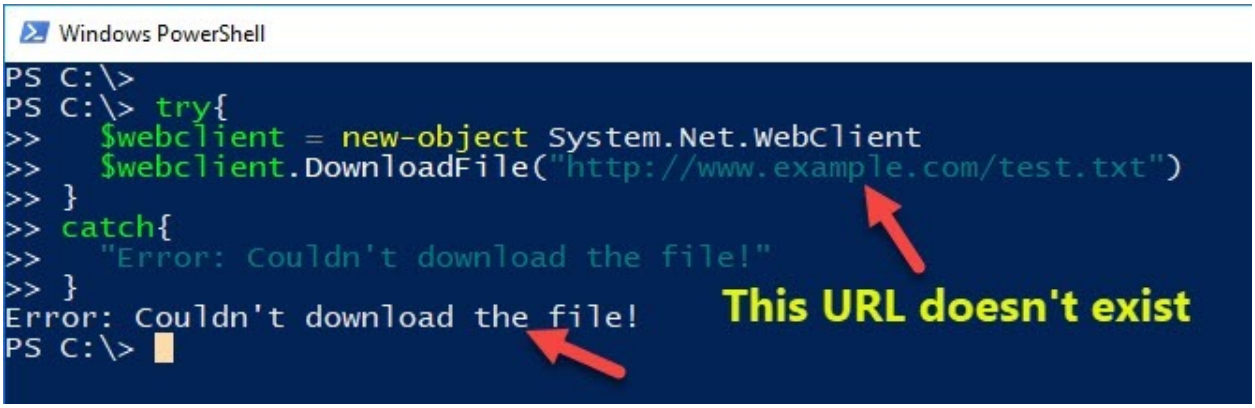
```
try:
  # statements
```

Catch and Except Block

When an error occurs in a `'try'` block then the PowerShell scripts search for a `'catch'` block to handle that error. A Catch block has a list of statements to handle the failure or to recover from the Error that was caught in `try` block.

```
try{
  $webclient = new-object System.Net.WebClient
  $webclient.DownloadFile("http://www.example.com/test.txt")
}
catch{
  "Error: Couldn't download the file!"
}
```

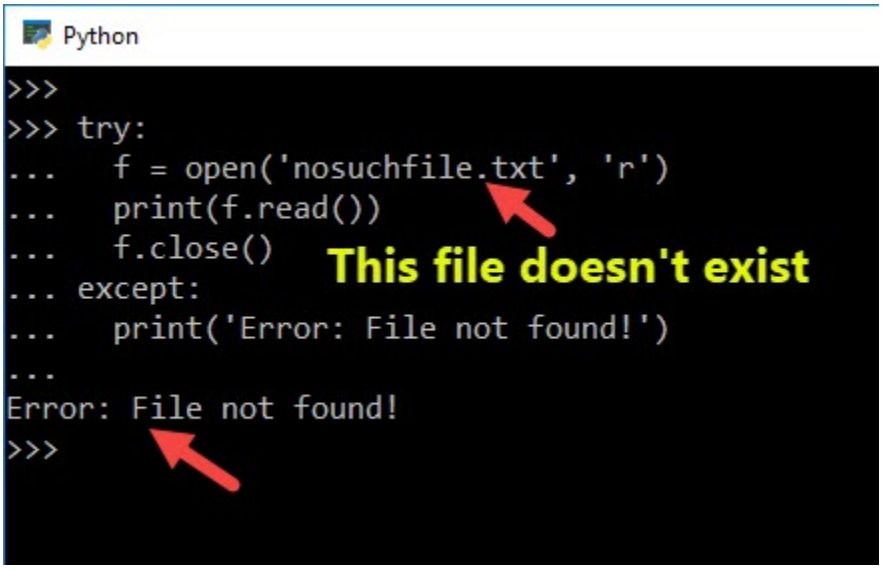
In the above example since no such URL exist the web client was unable to resolve the web address and throws an error. Since the `DownloadFile()` function is in `try` block which is getting monitored for any errors, as soon as the error was raised, it is then caught by `catch` block to handle it gracefully, which then returns a custom error message on your PowerShell console.



try..catch in PowerShell

Python on other hand has a `'except'` block to handle any errors which were thrown in `try` block

```python
try:
  f = open('nosuchfile.txt', 'r')
  print(f.read())
  f.close()
except:
  print('Error: File not found!')
```



try..except in Python

You can even define types of error a `'catch'` or `'except'` block can handle using built-in error types. The error type has to follow the `'catch'` or `'except'` keyword to specifically define a block that can handle a certain type of error.

In PowerShell `'Try'` statement can also have multiple `'Catch'` blocks that can handle multiple types of errors separately.

PowerShell Syntax:

```powershell
try{
  # statements
}
catch [ExceptionType1]{
  # Exception handler specific to <ExceptionType1> errors
}
catch [ExceptionType2]{
  # Exception handler specific to <ExceptionType2> errors
}
catch {
  # this catch block will handle any other error
```

```
  # that wasn't defined in previous catch blocks
}
```

Following example in PowerShell demonstrates that if a file doesn't exist and we try to read it in `'try'` block, then an error of type `[System.IO.FileNotFoundException]` is thrown, which will be caught by the `'catch'` block at line-5 because it specifically handling the errors of type: `FileNotFoundException`. Any errors that are not caught by one of the `catch` blocks which handle specific exceptions would be caught by the final `catch` block. This is a generic `catch` block capable of handling any errors other than `'FileNotFoundException'` and `'IOException'`.

```
try{
  $path = 'c:\temp\nosuchfile.txt'
  $data = [System.IO.File]::ReadAllText($path)
}
catch [System.IO.FileNotFoundException]{
  Write-Host "Error: Couldn't find file at path: $path"
}
catch [System.IO.IOException]{
  Write-Host "Error: I/O error with the file: $path"
}
catch{
  Write-Host "Error: Something went wrong"
}
```

```
PS C:\>
PS C:\> try{
  $path = 'c:\temp\nosuchfile.txt'
  $data = [System.IO.File]::ReadAllText($path)
}
catch [System.IO.FileNotFoundException]{
  Write-Host "Error: Couldn't find file at path: $path"
}
catch [System.IO.IOException]{
  Write-Host "Error: I/O error with the file: $path"
}
catch{
  Write-Host "Error: Something went wrong"
}
Error: Couldn't find file at path: c:\temp\nosuchfile.txt

PS C:\>
```

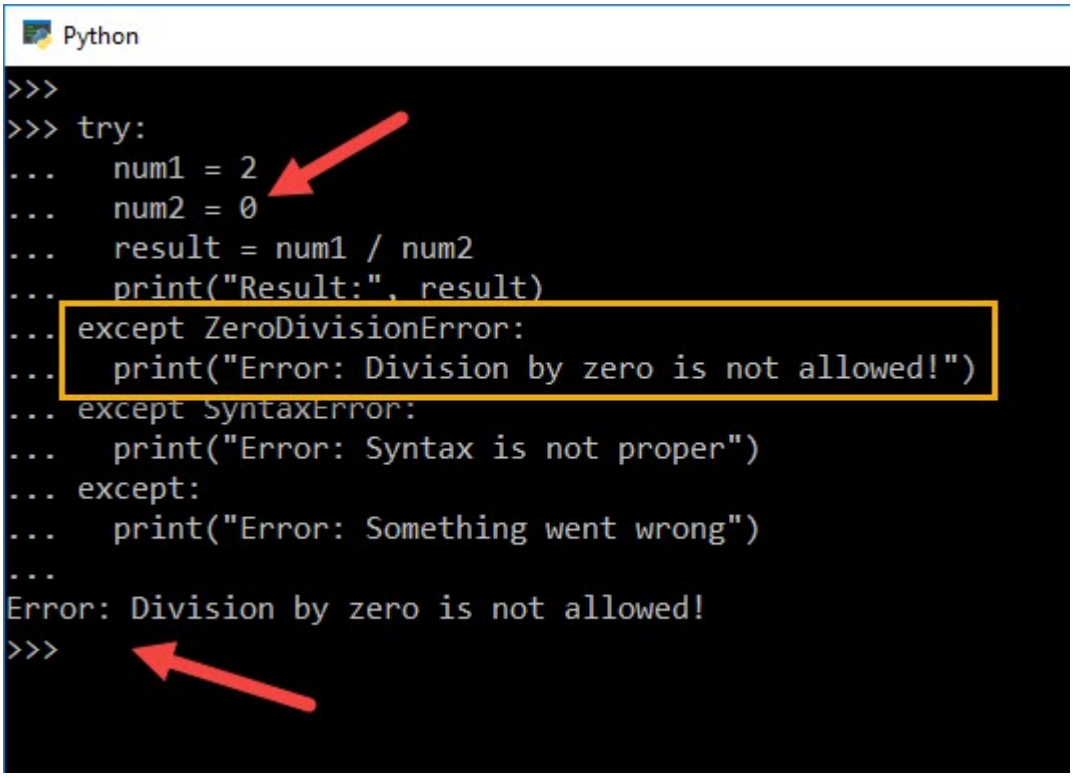
Similarly in Python you can also define multiple `'except'` blocks with exception types to handle various types of errors differently, with minor syntactical changes as demonstrated below:

Python Syntax:

```
try:
  # statements
except ExceptionType1:
  # Exception handler specific to <ExceptionType1> errors
except ExceptionType2:
  # Exception handler specific to <ExceptionType2> errors
except:
  # this catch block will handle any other error
  # that wasn't defined in previous catch blocks
```

In the following example we are trying to divide a number by Zero in Python which is expected to throw an `ZeroDivisionError`, but this error will be caught by the `'except'` block at line-6 because this is specifically handling the error of type:`'ZeroDivisionError'`.

```
try:
  num1 = 2
  num2 = 0
  result = num1 / num2
  print("Result:", result)
except ZeroDivisionError:
  print("Error: Division by zero is not allowed!")
except SyntaxError:
  print("Error: Syntax is not proper")
except:
  print("Error: Something went wrong")
```



Finally Block

A `Finally` block contains a set of statements that are finally executed no matter how the program ended, this is generally used to free any resources, that was used in the `try..catch` block of your program.

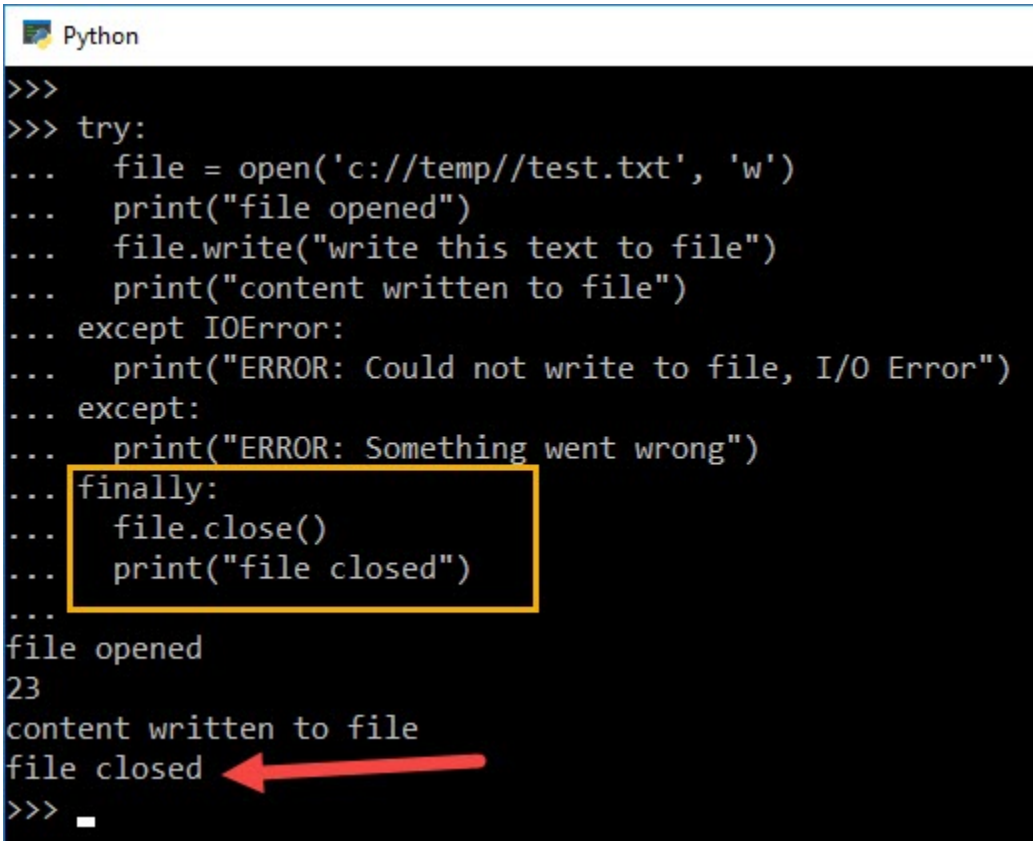
Basically, the `'Finally'` block contains the piece of program that is executed after completion of both the `Try` and `Catch` blocks regardless of whether an error occurred or not. The primary purpose of this block is clean resources like removing or disposing objects used by the program from memory to make it free.

PowerShell Syntax:

```
try{
  # statements to be monitored for errors
}
catch{
  # handle errors here
}
finally{
  # final tasks: these are executed no matter if an error is thrown or not
}
```

Let's take an example in PowerShell, in which we will attempt to open an instance of Internet Explorer web browser and navigate the URL in a `'try'` block. If we run into errors it will be caught by the `'catch'` block and after both `try` and `catch` blocks have finished executing then the `'finally'` block would be executed were we will close the instance of Internet Explorer.

```
try{
  $IE = New-Object -ComObject "InternetExplorer.Application"
  $Uri = "http://google.com/"
  $IE.Navigate($Uri)
  Write-Host "Launch Internet Explorer and Navigate URL:$Uri"
}
catch{
  Write-Host "Error: Something went wrong"
}
finally{
  $IE.Quit()
  Write-Host "Closing Internet Explorer"
}
```

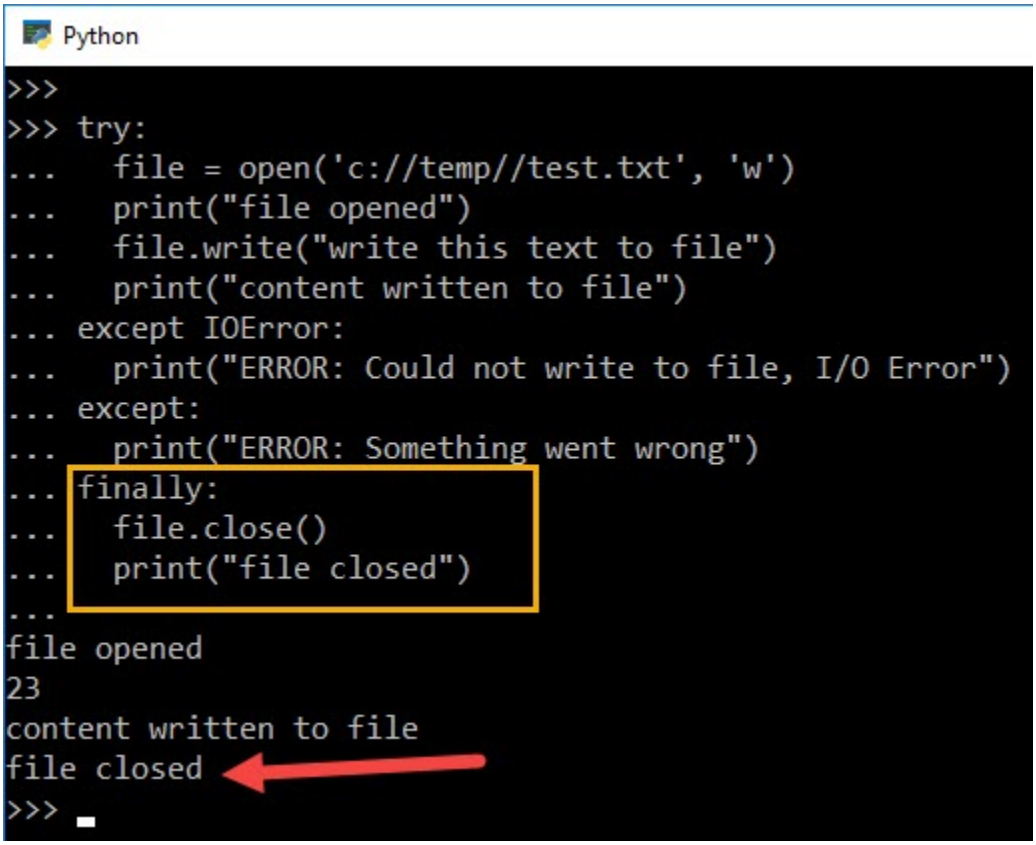


Python Syntax:

```
try:
  # statements to be monitored for errors
except:
  # handle errors here
finally:
  # final tasks: these are executed no matter if an error is thrown or not
```

Similarly, in Python we are opening a file and then we attempt to write a string to it inside a `'try'` block, any errors are thrown will be caught by one of the `'except'` blocks. At last after both the `try..except` blocks have finished execution, then the `'finally'` block will run to `close()` the file.

```
try:
  file = open('c://temp//test.txt', 'w')
  print("file opened")
  file.write("write this text to file")
  print("content written to file")
except IOError:
  print("ERROR: Could not write to file, I/O Error")
except:
  print("ERROR: Something went wrong")
finally:
  file.close()
  print("file closed")
```


Python
>>>
>>> try:
...   file = open('c://temp//test.txt', 'w')
...   print("file opened")
...   file.write("write this text to file")
...   print("content written to file")
... except IOError:
...   print("ERROR: Could not write to file, I/O Error")
... except:
...   print("ERROR: Something went wrong")
... finally:
...   file.close()
...   print("file closed")
...
file opened
23
content written to file
file closed
>>>

Chapter 20 - Built-In Functions

It would be fair enough to say that functions that are in built-in in a programming language are very useful and can be a huge advantage in day to day tasks because you don't have to write them yourself. Python has a lot of functions built into it that are always available to you. Similarly, PowerShell has access to .Net classes that brings some useful functions to the shell as well. So in this chapter, we are going to cover various built-in functions provided by PowerShell and Python.

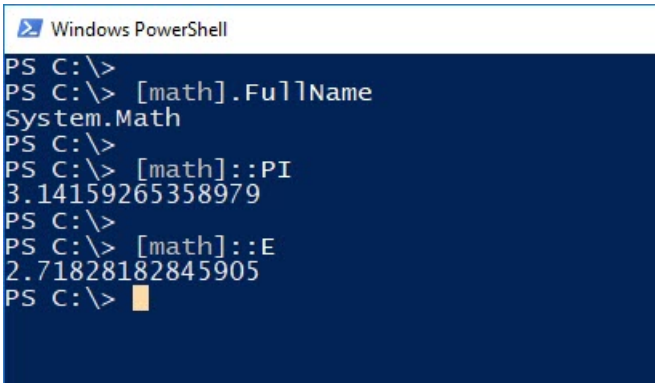
Basic Mathematical Operations

You'll come across various day-to-day tasks where it is required to perform basic mathematical operations, let us look into them one at a time:

Constants: PI and E

Mathematical constant `Pi = 3.141592` and Euler's number: `e = 2.718281` can be accessed in PowerShell using the `System.Math` class which has static properties holding values of these constants as shown in the following example:

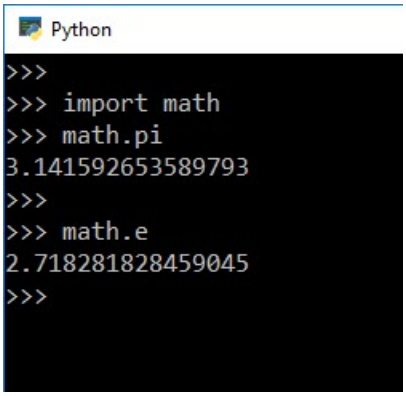
```
1  [math]::PI
2  [math]::E
```



Likewise, Python has a `math` module that can be utilized to access the values of these constants.

Python

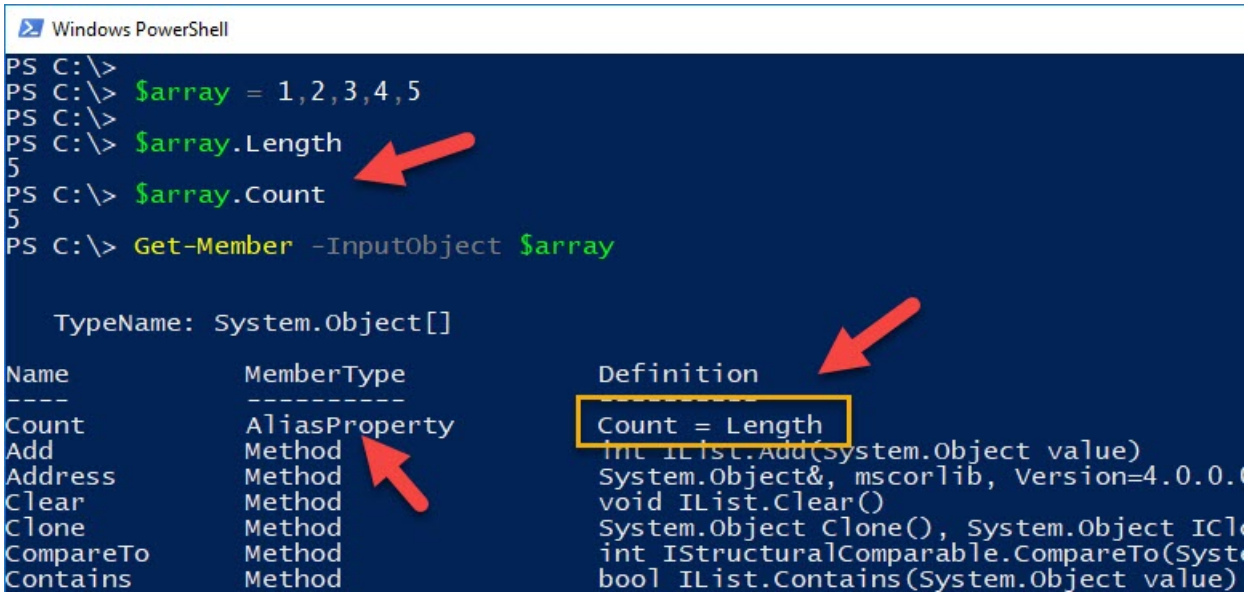
```
import math
math.pi
math.e
```



Count and Sum: len() and sum()

PowerShell by default adds a property: `length` to containers like an array, which also has an alias named `count` that can be utilized to get the count of items in an array.

```
$array = 1,2,3,4,5
$array.Length

# alternatively
$array.Count
```

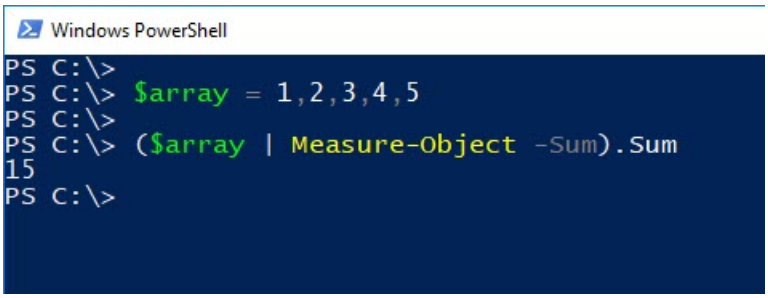


Python on other hand has a built-in function `len()` that can return the number of items in a container.

```
len([1,2,3])
```

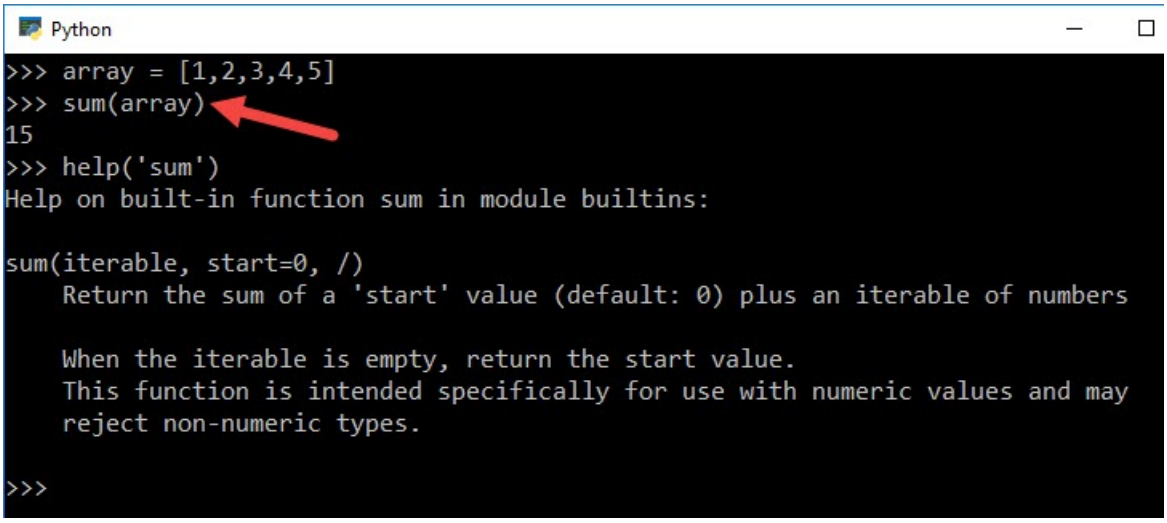
Adding values together in PowerShell is as easy as piping the values to `Measure-Object` cmdlet with a `-Sum` switch

```
$array = 1,2,3,4,5
($array | Measure-Object -Sum).Sum
```



Whereas in Python utilizes `sum()` function to add items of a list and return the sum.

```
array = [1,2,3,4,5]
sum(array)
```



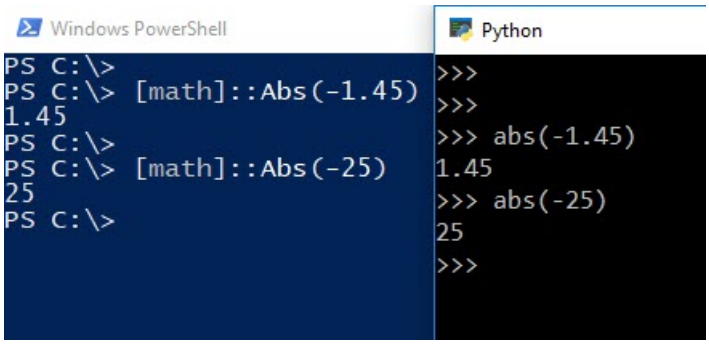
Absolute and Rounded Value: abs() and round()

Python has an inbuilt function: `abs()` that can return the absolute value of a number, which is an integer or a floating point number.

```
abs(-1.45)
abs(-25)
```

Likewise, in PowerShell you can use the `[math]::Abs()` method of the `System.Math` class to get the absolute values of the number passed as an argument.

```
[math]::Abs(-1.45)
[math]::Abs(-25)
```



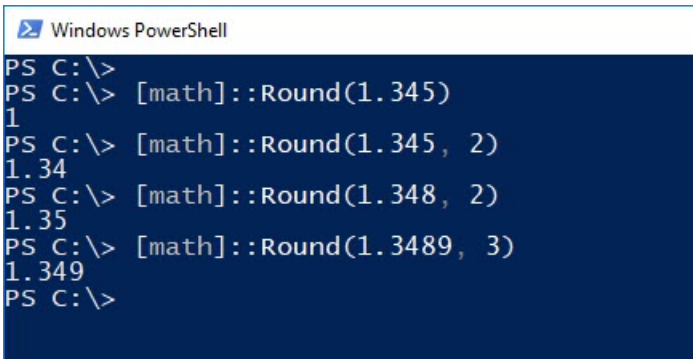
`'System.Math'` class also provide a `Round()` method, that will return rounded floating point number: `'x'` up to decimal digits: `'n'`, if `'n'` is omitted then default value: `0` is taken.

Syntax:

```
[math]::Round(x,n)
```

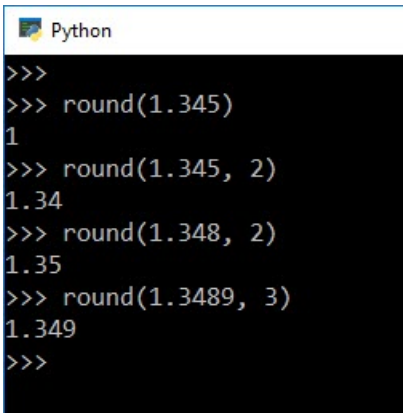
Example:

```
[math]::Round(1.345)
[math]::Round(1.345, 2)
[math]::Round(1.348, 2)
[math]::Round(1.3489, 3)
```



Python on other hand has a built-in function: `round()` to return the rounded floating point values.

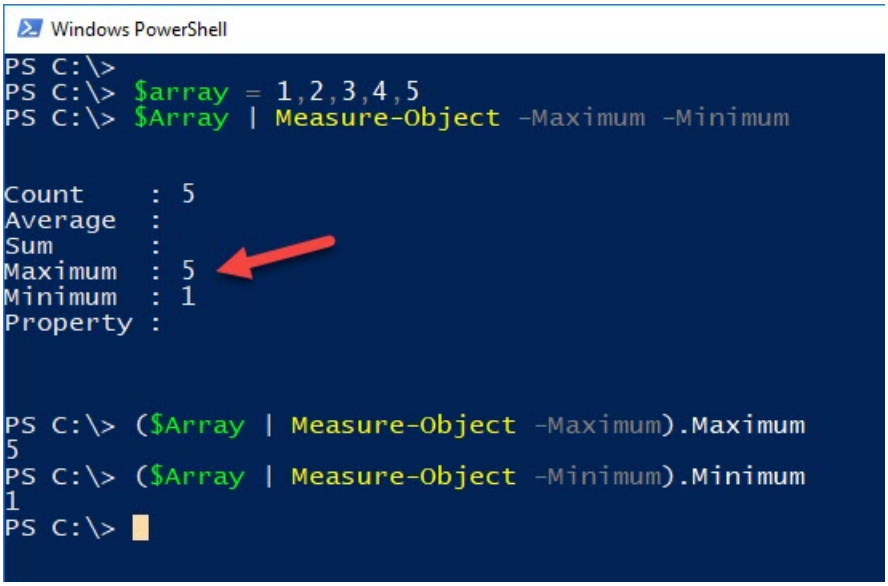
```
round(1.345)
round(1.345, 2)
round(1.348, 2)
round(1.3489, 3)
```



Maximum and Minimum: max() and min()

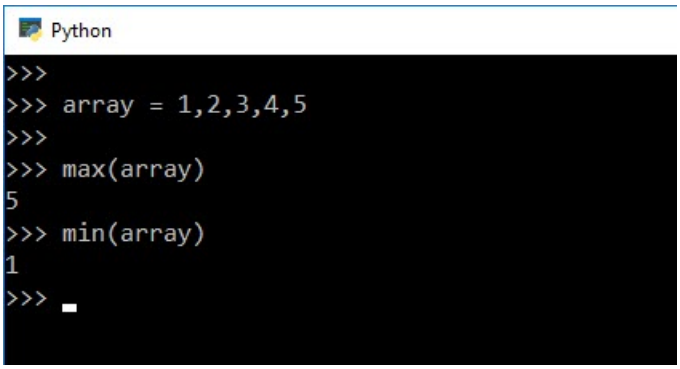
Finding maximum and minimum number in PowerShell is achieved by using the `Measure-Object` cmdlet with `-Max` and `-Min` switches on an array of items.

```
$array = 1,2,3,4,5
$Array | Measure-Object -Maximum -Minimum

# finding maximum number
($Array | Measure-Object -Maximum).Maximum

# finding minimum number
($Array | Measure-Object -Minimum).Minimum
```



Whereas, Python has built-in functions: `max()` and `min()` to find largest or smallest items in a list.

```
array = 1,2,3,4,5

# finding maximum number
max(array)

# finding minimum number
min(array)
```



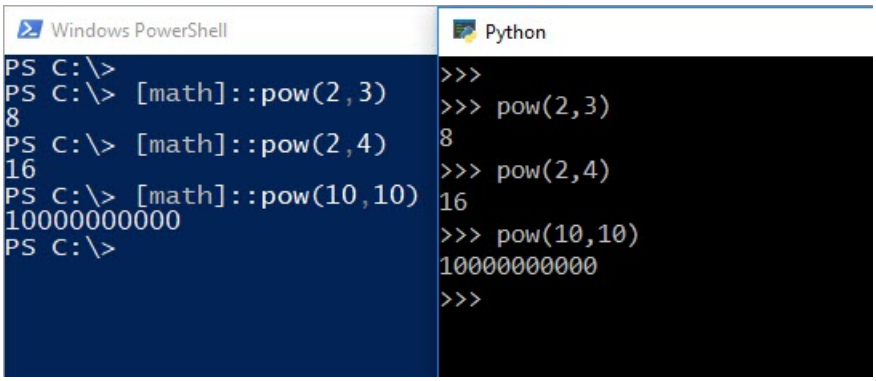
Powers and Square Roots: pow() and sqrt()

PowerShell can access the `Pow(x, y)` method of `'System.Math'` class, to return `'x to the power y'`. Both the arguments passed to this function are numerical values.

```
[math]::pow(2,3)
[math]::pow(2,4)
[math]::pow(10,10)
```

Python has a built-in function `pow(x, y)` to get the x to the power y.

```
pow(2,3)
pow(2,4)
pow(10,10)
```

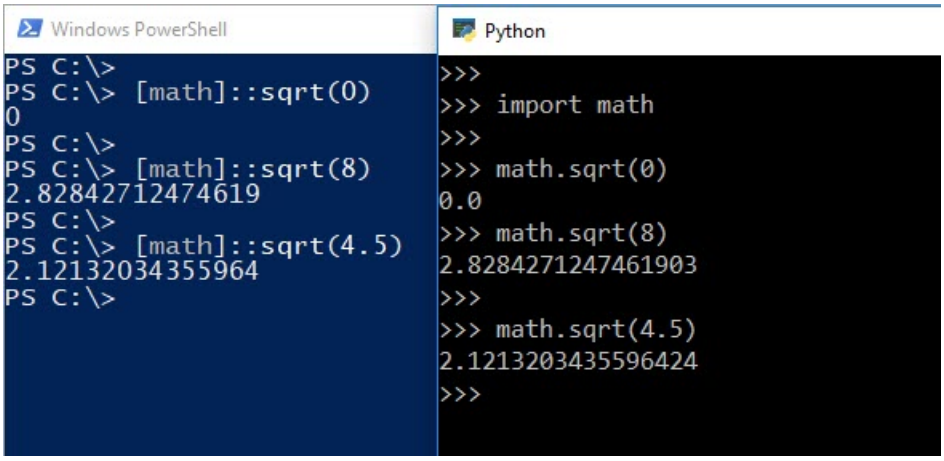


In PowerShell we can use the `sqrt()` function of `System.Math` class to calculate the square roots.

```
[math]::sqrt(0)
[math]::sqrt(8)
[math]::sqrt(4.5)
```

Whereas, in Python, you have to first import the `'math'` module, then access the `sqrt()` function using the (`.`) dot operator, to return the square roots.

```
import math

math.sqrt(0)
math.sqrt(8)
math.sqrt(4.5)
```



Trigonometric Functions: sin(), cos() and tan()

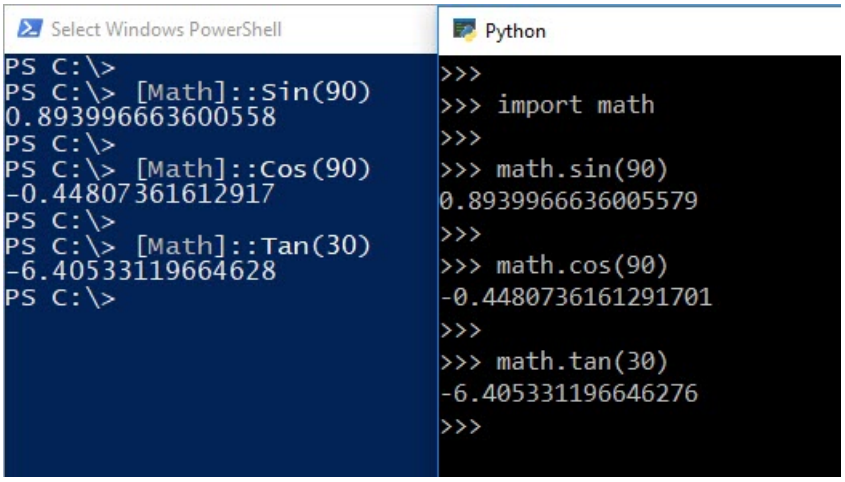
Both the languages have inbuilt Trigonometric functions to calculate Sine, Cosine, Tangent of an angle as a numerical value.

PowerShell:

```
[Math]::Sin(90)
[Math]::Cos(90)
[Math]::Tan(30)
```

Python:

```
import math

math.sin(90)
math.cos(90)
math.tan(30)
```



Logarithm: log() and log10()

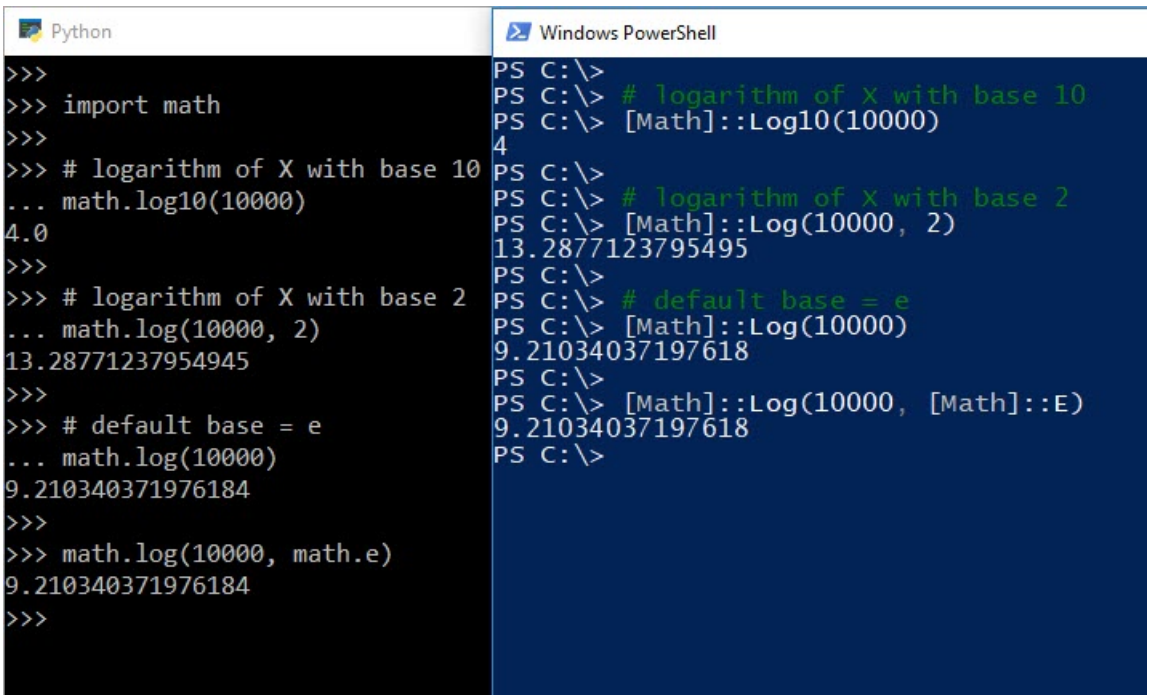
Logarithm is a function used to reverse the operation of exponentiation. For example, when the fourth power of 10 equals 10000, then the logarithm of 10000 with respect to base 10 is 4. In PowerShell you can use the `Log10()` function from the `System.Math` class to return the logarithm of 10000 for base 10, similarly `'math'` library in Python has a `log10()` function. Both the languages also provide a `log()` function that can return the logarithm of x to the given base and if the base is not specified, then the natural logarithm (base `'e'`) of x is returned.

PowerShell:

```
# logarithm of X with base 10
[Math]::Log10(10000)

# logarithm of X with base 2
[Math]::Log(10000, 2)

# default base = e
[Math]::Log(10000)

[Math]::Log(10000, [Math]::E)
```

Python:

```python
import math

# logarithm of X with base 10
math.log10(10000)

# logarithm of X with base 2
math.log(10000, 2)

# default base = e
math.log(10000)

math.log(10000, math.e)
```

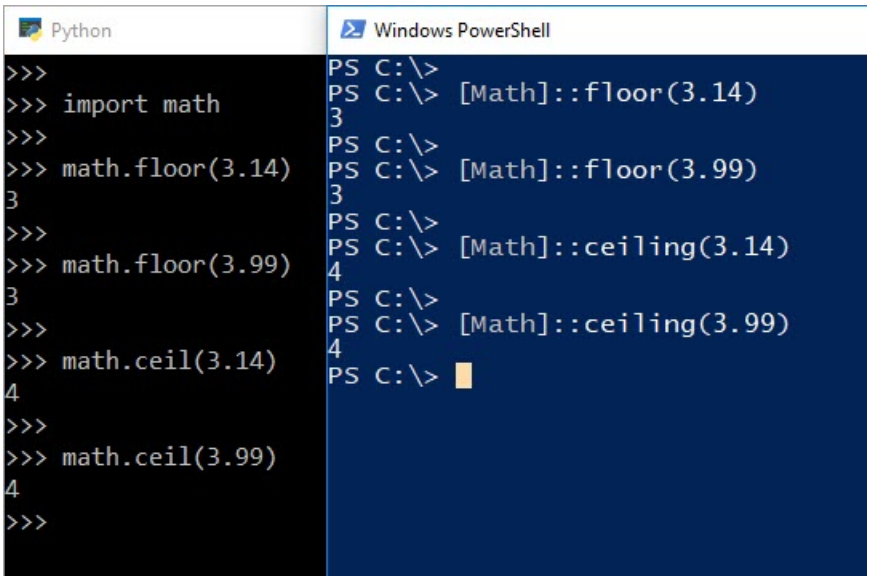


Floor and Ceiling: floor() and ceil()

In Python and PowerShell, the `floor(x)` function returns the floor of `'x'` as an Integral, which is the largest integer less-than equals-to x (`<= x`). On other hand Python has a `ceil(x)` function that returns the ceiling of `'x'` as an Integral, which is the smallest integer greater-than equals-to x (`>= x`). But in PowerShell this function is called `Ceiling(x)` providing the same functionality.

```powershell
[Math]::floor(3.14)
[Math]::floor(3.99)
[Math]::ceiling(3.14)
[Math]::ceiling(3.99)
```

```python
import math

math.floor(3.14)
math.floor(3.99)
math.ceil(3.14)
math.ceil(3.99)
```



Selection and Arrangement

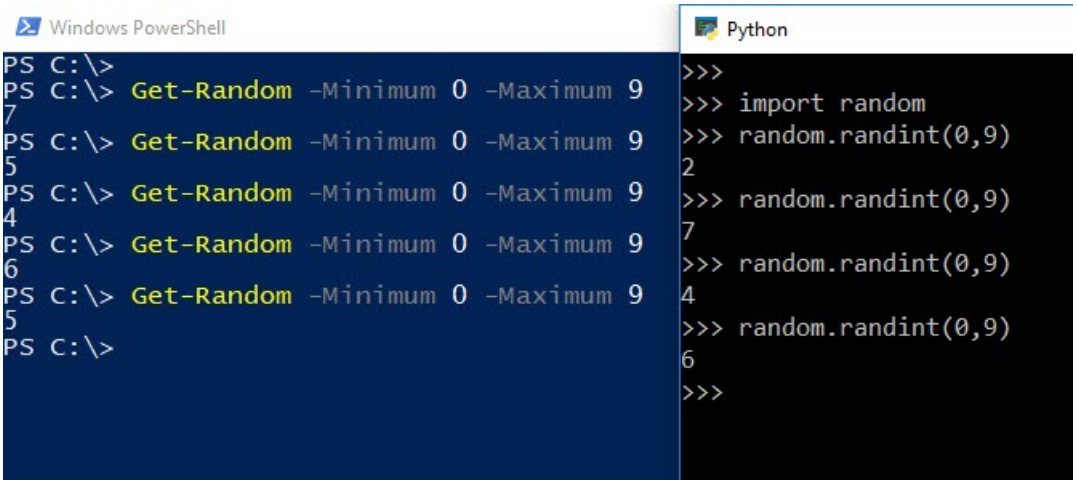
Random Selection

PowerShell has a `Get-Random` cmdlet to return a random number, where you can define the maximum and minimum possible number within a range.

```powershell
Get-Random -Minimum 0 -Maximum 9
```

Python has a `'random'` module which provides a `randint()` method that can return a random number from a range of integers passed as the argument.

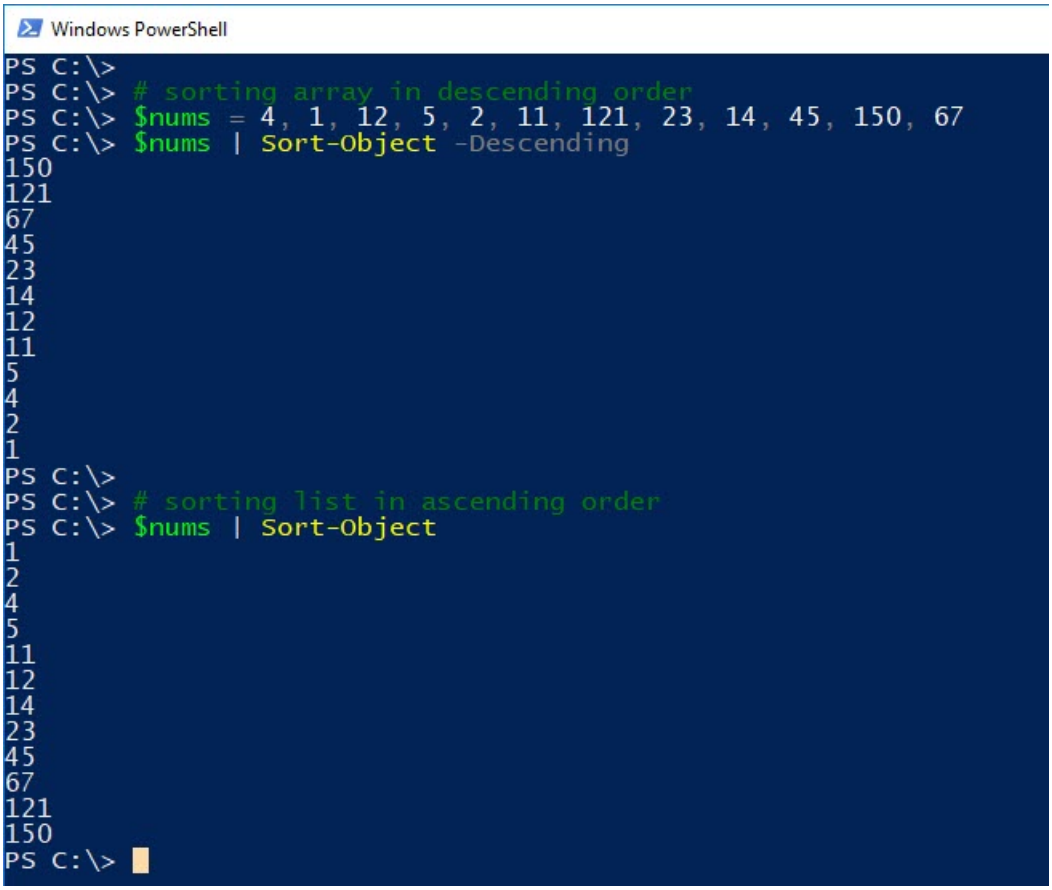
```
import random
random.randint(0,9)
```



Sorting in Ascending and Descending Order: sort()

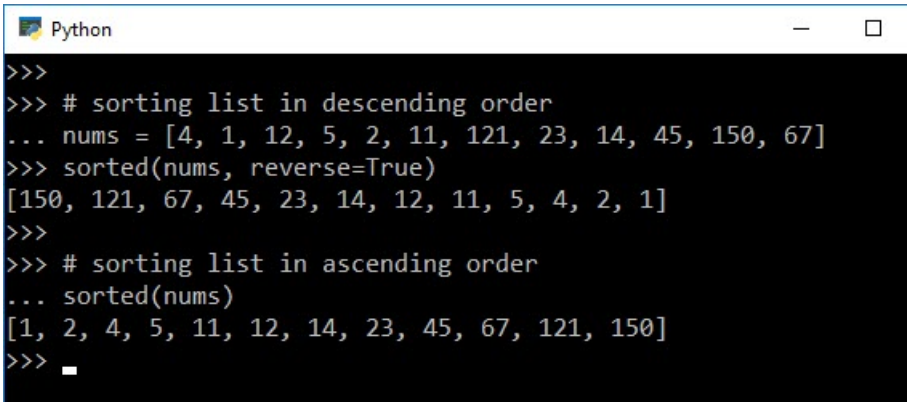
PowerShell has `Sort-Object` cmdlet that can sort arrays into sorted lists in descending order using the `-Descending` switch, and by default if this switch is not specified it sorts the items in ascending order.

```
# sorting array in descending order
$nums = 4, 1, 12, 5, 2, 11, 121, 23, 14, 45, 150, 67
$nums | Sort-Object -Descending

# sorting list in ascending order
$nums | Sort-Object
```



Python has a built-in function: `sorted()` that can sort lists to descending order, when boolean `True` is passed to the named argument `reverse`, by default the `sorted()` function returns list in ascending order.

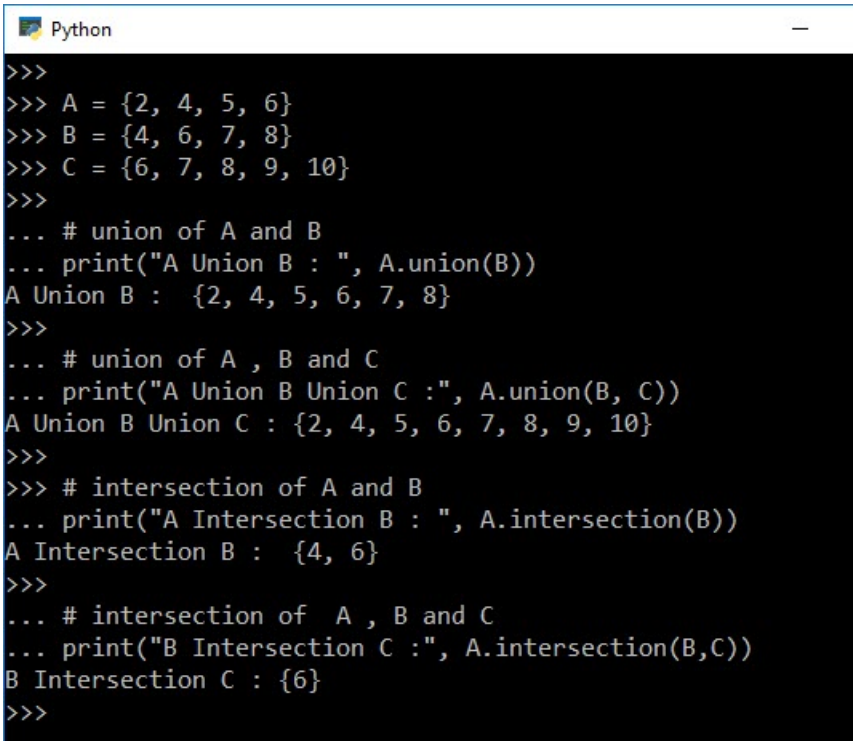
```
1  # sorting list in descending order
2  nums = [4, 1, 12, 5, 2, 11, 121, 23, 14, 45, 150, 67]
3  sorted(nums, reverse=True)
4
5  # sorting list in ascending order
6  sorted(nums)
```



Union and Intersection

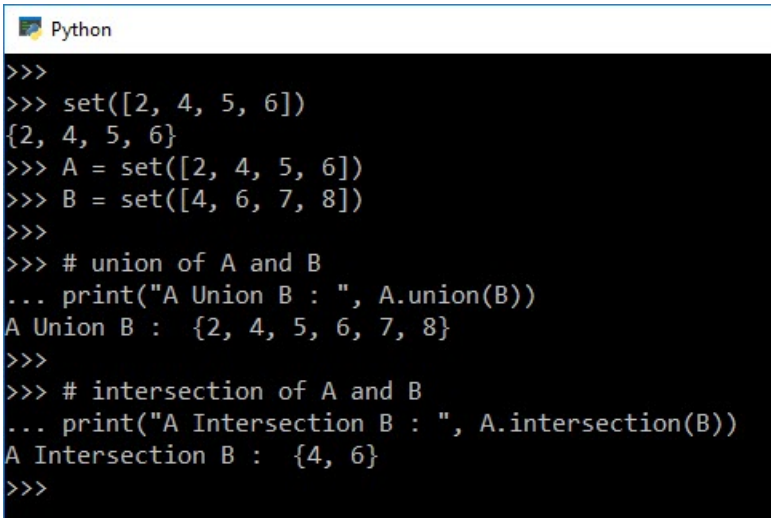
The union of two or more sets is the set of all distinct elements present in all the sets, and Intersection of given sets consists of all the elements which are common in all the sets. In Python, you can perform these operations on a Set which is a list of unique items enclosed in curly braces `{ }`, all Set objects have inbuilt functions `union()` or `intersection()`.

```
A = {2, 4, 5, 6}
B = {4, 6, 7, 8}
C = {6, 7, 8, 9, 10}

# union of A and B
print("A Union B : ", A.union(B))

# union of A , B and C
print("A Union B Union C :", A.union(B, C))

# intersection of A and B
print("A Intersection B : ", A.intersection(B))

# intersection of  A , B and C
print("B Intersection C :", A.intersection(B,C))
```



In case you have Python Lists within square brackets `[ ]` instead of Sets with curly brackets `{ }` you can convert such lists to sets using the `'set()'` built in function. To know more run the following command from Python console: `help(set)`

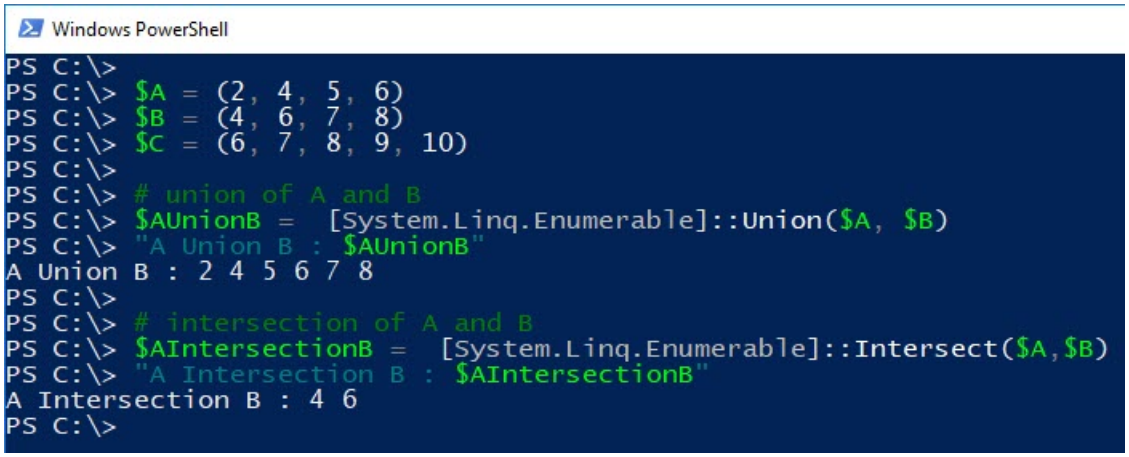
```
A = set([2, 4, 5, 6])
B = set([4, 6, 7, 8])

# union of A and B
print("A Union B : ", A.union(B))

# intersection of A and B
print("A Intersection B : ", A.intersection(B))
```



Converting Lists to Sets to get Union & Intersection

In PowerShell, we can utilize .Net `[System.Linq.Enumerable]` class, that provide static methods: `Union()` and `Intersect()` which can be used to return Union and Intersection from lists of items. LINQ stands for Language-Integrated Query, that defines a set of general purpose standard query operators that allow traversal and filtering operations.

```
$A = (2, 4, 5, 6)
$B = (4, 6, 7, 8)
$C = (6, 7, 8, 9, 10)

# union of A and B
$AUnionB =  [System.Linq.Enumerable]::Union($A, $B)
"A Union B : $AUnionB"

# intersection of A and B
$AIntersectionB =  [System.Linq.Enumerable]::Intersect($A,$B)
"A Intersection B : $AIntersectionB"
```



Dynamic Execution of Code

With Python's `exec()` function, you can execute code from a string or generate a piece of code dynamically during the running of a program.

```
prog = 'print("The sum of 5 and 10 is", (5+10))'
exec(prog)
```

PowerShell on other hand has Script blocks, which can be created dynamically during a running program and can also be executed from a string. To know more about the Script Blocks run the following command from PowerShell console: `Get-Help about_Script_Blocks`

```
$prog = {"The sum of 5 and 10 is $(5+10)"}
$prog.invoke()

# alternatively
& $prog

# alternatively
[ScriptBlock]::Create("The sum of 5 and 10 is $(5+10)")
```