

The background of the book cover features a photograph of a man from behind, sitting at a desk and looking at multiple computer monitors displaying code. Overlaid on this image is a grid of white lines forming squares. Inside many of these squares are various icons related to technology and automation, such as a cloud with binary code, a gear, a magnifying glass over code, a laptop with binary code, a shield with a checkmark, and several padlocks. Some squares also contain blue or white dots. The overall color scheme is dark blue and black with white and light blue accents.

FOREWORD WRITTEN BY ORIN THOMAS

MODERN IT AUTOMATION
WITH POWERSHELL
FIRST EDITION

EDITED BY

MICHAEL ZANATTA
BILL KINDLE

STEVEN JUDD
NICHOLAS BISSELL

The background of the book cover features a photograph of a man from behind, sitting at a desk and looking at multiple computer monitors displaying code. Overlaid on this image is a grid of white squares containing various blue and purple icons related to technology and automation. These icons include binary code (0s and 1s), a cloud with circuit lines, a gear, a magnifying glass over a code editor window, a laptop showing code, a document with a checkmark, a padlock, curly braces {}, and a stylized face or mask. The overall aesthetic is dark and technical.

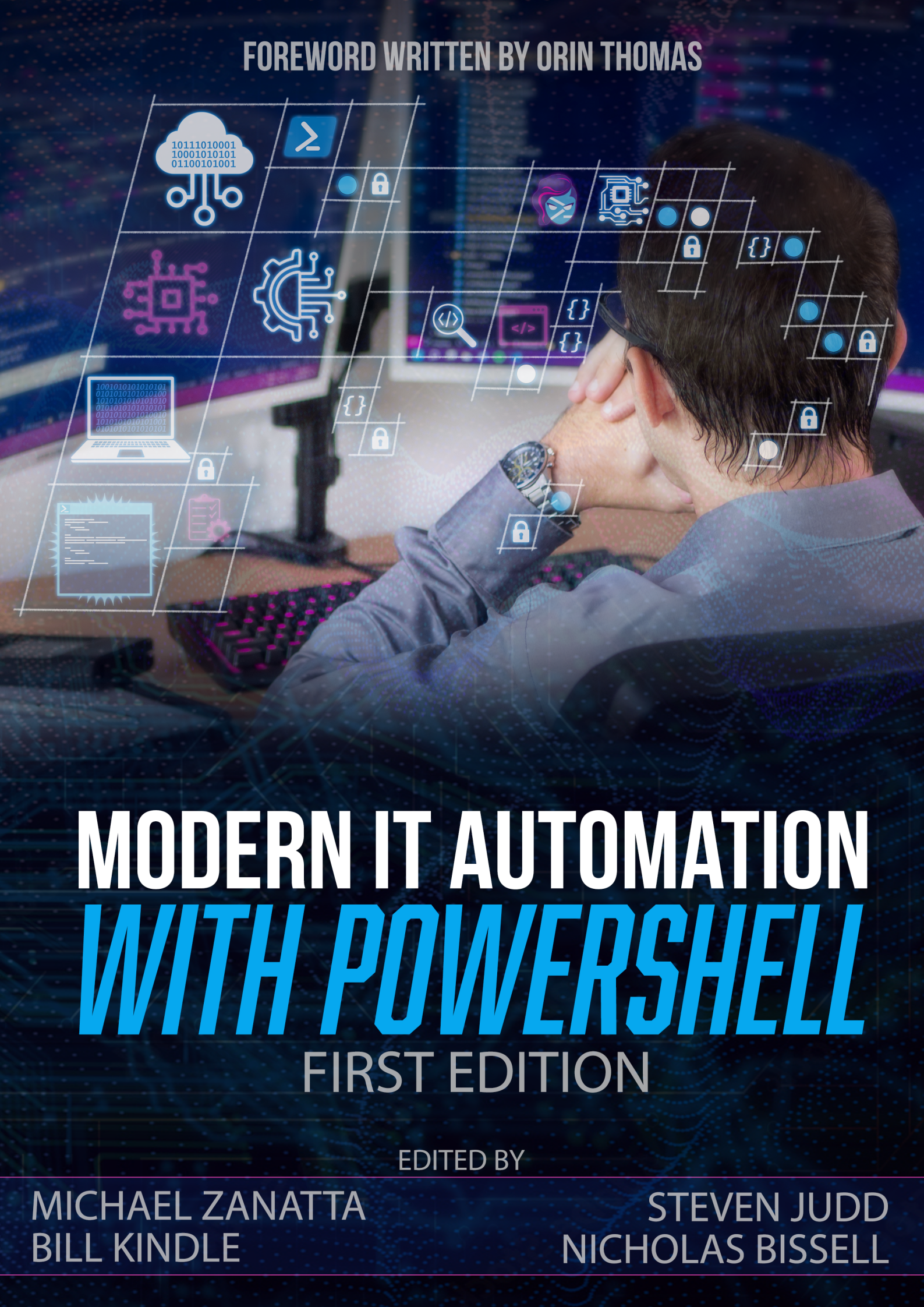
FOREWORD WRITTEN BY ORIN THOMAS

MODERN IT AUTOMATION
WITH POWERSHELL
FIRST EDITION

EDITED BY

MICHAEL ZANATTA
BILL KINDLE

STEVEN JUDD
NICHOLAS BISSELL

[illegible]The book cover features a background image of a person sitting at a desk, looking at a computer screen. Overlaid on this image is a grid of various icons representing IT and automation concepts, including a cloud with binary code, a gear, a magnifying glass over code, a laptop with binary code, a shield, a padlock, a circuit board, a brain, a code editor window, and a checklist. The title 'MODERN IT AUTOMATION' is in white, 'WITH POWERSHELL' is in large blue letters, and 'FIRST EDITION' is in white. The authors' names are at the bottom in white.

FOREWORD WRITTEN BY ORIN THOMAS

MODERN IT AUTOMATION
WITH POWERSHELL
FIRST EDITION

EDITED BY

MICHAEL ZANATTA
BILL KINDLE

STEVEN JUDD
NICHOLAS BISSELL

[illegible]

Modern IT Automation with PowerShell

Modern Automation with PowerShell

The DevOps Collective, Inc. and Michael Zanatta

This book is for sale at <http://leanpub.com/modernautomationwithpowershell>

This version was published on 2022-09-20



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2022 The DevOps Collective, Inc. All Rights Reserved

Tweet This Book!

Please help The DevOps Collective, Inc. and Michael Zanatta by spreading the word about this book on [Twitter](#)!

The suggested tweet for this book is:

I'm supporting the future of our field and just bought a copy of The [#pwshconftextbook](#) First Edition!

The suggested hashtag for this book is [#pwshconftextbook](#).

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

[#pwshconftextbook](#)

Contents

Foreword	i
Contributors	ii
Alain Tanguy	ii
Allen Chin	ii
Amy Zanatta	ii
Bill Kindle	iii
C.J. Zuk	iii
Chad Miars	iii
Christian Coventry	iii
Felipe Binotto	iv
Greg Onstot	iv
James Petty	iv
Joe Houghes	v
John Hermes	v
Jordan Borean	v
Kevin Laux	vi
Kieran Jacobsen	vi
Kirill Nikolaev	vi
Martha Clancy	vii
Matt Corr	vii
Michael B. Smith	vii
Michael Lotter	viii
Michael Zanatta	viii
Nicholas Bissell	viii
Rob Derickson	ix
Steven Judd	ix
Wes Stahler	ix
Acknowledgements	x
Disclaimer	xi
Introduction	xii
About OnRamp	xii
Prerequisites	xiii
A Note on Code Listings	xiii
Feedback	xiv

Using Regexes	xv
Regex 101	xvi
First Principles and Limitations	xvi
Wildcard Patterns vs. Regexes	xvii
Differences Between PowerShell Regexes and Others	xvii
Getting Started	xviii
Character Classes	xix
Custom Character Classes	xxi
Quantifiers	xxii
Character Escape Sequences	xxiv
Anchors (<i>Zero-Width Assertions</i>)	xxv
Captures	xxvi
Visualizing Captures	xxvii

Foreword

By Orin Thomas

Organizations adopt information technology to solve a set of problems. The problems could be as simple as “how do we keep track of customer orders?” or more complicated ones involving the analysis of data to determine patterns that might provide some new insight that leads to a business advantage. Organizations will choose a technology not just because they think it is fun and cool, but because they can use it to solve a problem that they really need to be solved to accomplish an organizational objective. The creators of information technologies often have a set of problems in mind when building those technologies. It might be “how do I simplify the process of creating and presenting slides at conferences” or “how can we better automate administrative tasks on Microsoft platforms”. Every system, application, or programming language has a particular set of tasks it was designed to do very well because the people that created it needed to scratch an itch that the currently available tools did not adequately address.

Over time creators and developers add and refine features to their products or tools because those features to allow users of the technology to solve additional or more complex problems that are considered important. But the key is understanding that to the creator and developer there is a set of problems that are “in scope” that they envisage their project solving and a lot that are beyond that scope that it was never intended to address. William Gibson, author of one of the earliest and most important cyberpunk novels said “The street finds its own uses for things”. One of the interpretations of this is that users often find uses for products that go way beyond what the original developers of that product envisaged it being used for. Good products are useful for things that the original creators never imagined. No product can do everything, but with a bit of creativity, many products can do things that are a complete surprise to other users and even the original creators of that project.

Technical communities are groups of people that are enthusiastic about specific systems, applications, technologies, and languages. These communities spend a great amount of time not only sharing how to be more proficient in doing something that the technology that they are interested in was designed to do, but also sharing all of the amazing things that the technology does that no one could have imagined. Jeffrey Snover often remarks how surprised he is at all the ingenious things people find that they can do with PowerShell that he never conceived of it being used for.

And that’s what this book is about—sharing with the reader a collection of fascinating things that you can do with PowerShell. Not only things that you already do that you might be able to do in a much more efficient or elegant way, but a collection of tasks that you can do with PowerShell that exceed what you and perhaps even Jeffrey Snover conceived the language was possible of accomplishing.

Contributors

This section includes the names and biographies of the authors and editors of this project in alphabetical order.

Alain Tanguy

Role: Author

Alain has been an avid PowerShell user throughout his IT career, automating and solving many problems. Now working as an IT Engineer, he is using his knowledge to answer IT Infrastructure Challenges. He enjoys pizza, pixel art, funky music, and naps. Alain is reachable on Twitter at [@Alain__Tanguy](https://twitter.com/Alain__Tanguy)¹ and [LinkedIn](https://www.linkedin.com/in/alaintanguy)².

Allen Chin

Role: Linguistic Editor

Allen first learned and made use of PowerShell in 2018 while working as technical support to maintain and improve existing scripts. A few years later, he is now an application development analyst and the main contact for applications, reports, and automation.

Amy Zanatta

Role: Cover Artist

Amy is a Video Editor, Motion Graphics Designer and Personal Stylist, working at the Nine Network Australia. Amy posts regularly on her YouTube Channel [StyleWithinGrace](https://www.youtube.com/channel/UCCF1px3YSkNKB6F0HtySI9g)³, with Australian fashion ideas and inspirations. You can follow her on [Instagram](https://www.instagram.com/stylewithgrace/)⁴ and Twitter [@StyleGraceAmy](https://www.twitter.com/StyleGraceAmy/)⁵.

¹https://twitter.com/Alain__Tanguy

²<https://www.linkedin.com/in/alaintanguy>

³<https://www.youtube.com/channel/UCCF1px3YSkNKB6F0HtySI9g>

⁴<https://www.instagram.com/stylewithgrace/>

⁵<https://www.twitter.com/StyleGraceAmy/>

Bill Kindle

Role: Senior Editor

Husband to a wonderful woman that he doesn't deserve, and the father of two adorable children. Bill is a former career Systems Administrator turned Cyber Security Engineer currently working for [Corsica Technologies](https://corsicatech.com)⁶. Bill was an author for The PowerShell Conference Book Volume 2 and an editor for Volume 3. His role focuses on automation engineering and supporting a Security Operations Center. Bill has a passion for helping others in IT and occasionally does presentations for [@FortWayneVMUG](https://twitter.com/FortWayneVMUG)⁷. You can find some of Bill's work at [AdamTheAutomator](https://adamtheautomator.com)⁸ and [TechSnips LLC](https://techsnips.io)⁹.

C.J. Zuk

Role: Linguistic Editor

C.J. Zuk is a current college student and works as a federal government contractor in the Washington, D.C. Metropolitan Area. She began to use PowerShell in high school as her preferred scripting language because of its cross-platform compatibility. C.J. likes to spend her time with her fiancée, cats, and volunteering at her synagogue. You can reach her at cj.zuk@outlook.com and on [LinkedIn](https://www.linkedin.com/in/cj-zuk/)¹⁰.

Chad Miars

Role: Linguistic Editor

Chad's formal training is in mechanical engineering, but his current role is Director of Business Ops at [Ascent Inc.](https://ascenthvac.com)¹¹. He enjoys using PowerShell to connect and automate all parts of the business. You can find Chad on [LinkedIn](https://www.linkedin.com/in/chad-miars-6489a2122/)¹².

Christian Coventry

Role: Linguistic Editor

Christian is a recent graduate, and it was his tertiary studies that introduced him to PowerShell. He currently works as a Technical Officer with the [Queensland Department of Education](https://education.qld.gov.au/)¹³. Christian was an editor for The PowerShell Conference Book Volume 3. You can find him on [LinkedIn](https://au.linkedin.com/in/christian-coventry-9a0031157)¹⁴.

⁶<https://corsicatech.com>

⁷<https://twitter.com/FortWayneVMUG>

⁸<https://adamtheautomator.com>

⁹<https://techsnips.io>

¹⁰<https://www.linkedin.com/in/cj-zuk/>

¹¹<https://ascenthvac.com>

¹²<https://www.linkedin.com/in/chad-miars-6489a2122/>

¹³<https://education.qld.gov.au/>

¹⁴<https://au.linkedin.com/in/christian-coventry-9a0031157>

Felipe Binotto

Role: Author

Felipe is a specialist in Microsoft technologies and has worked in various roles in the IT industry for the last 13 years. Currently, he works as a Senior Customer Engineer for the Customer Success Unit at Microsoft Australia. In this role, his focus is on Azure infrastructure, security, and automation. Felipe has contributed to various PowerShell forums and Microsoft Docs, and he currently blogs at [Azure Gear](https://azuregear.com)¹⁵. You can follow him on Twitter at [@felipebinotto](https://twitter.com/felipebinotto)¹⁶ or on [LinkedIn](https://www.linkedin.com/in/felipebinotto/)¹⁷.

Greg Onstot

Role: Author

Husband, Father, Contributor to the PowerShell Conference Book Vol. 2. Greg has been working in IT Infrastructure and Cybersecurity Engineering for over 20 years. PowerShell is an integral part of automating that work.

James Petty

Role: DevOps Collective Liaison

James currently serves as the CEO of the DevOps Collective Inc., a nonprofit working in the technology education space. He helps manage a \$1M+ annual budget that includes multiple conferences and PowerShell Saturdays events across the US. The nonprofit focuses on PowerShell, automation, and DevOps and runs numerous free online resources, including PowerShell.org. He is also a co-organizer and co-founder of the Chattanooga PowerShell UserGroup (established in September 2016). James is also a recipient of the Microsoft MVP award in Cloud and Datacenter Management. He currently lives in the beautiful Chattanooga, Tennessee area with his amazing wife. James' passion lies with automation using PowerShell and all things related to Windows Server. He has almost a decade of experience as an infrastructure admin for a large enterprise, helping manage thousands of users and machines. He knows a broad range of products, including patch management, Active Directory, Group Policy, and the Windows Server operating system.

¹⁵<https://azuregear.com>

¹⁶<https://twitter.com/felipebinotto>

¹⁷<https://www.linkedin.com/in/felipebinotto/>

Joe Houghes

Role: Technical Editor

Husband, Father, Community Geek. Joe Houghes is a co-leader of [@ATXPowerShell](https://twitter.com/ATXPowerShell)¹⁸ and [@AustinVMUG](https://twitter.com/AustinVMUG)¹⁹ user groups in Texas and a member of the [@vBrownBag](https://twitter.com/vbrownbag)²⁰ crew. He is currently a Solutions Architect for Veeam, focused on automation & integration. Joe spends most of his time working within VMware environments when he is not active in planning or hosting community events. You can find Joe on Twitter, [@jhoughes](https://twitter.com/jhoughes)²¹, or his [blog](https://www.fullstackgeek.net/)²².

John Hermes

Role: Linguistic Editor

John is an agile software developer and systems engineer with a career focus on security and resilience. He frequently develops PowerShell modules supporting datacenter management, legacy systems, and cloud service integration. He is also an unabashed Unix greybeard who still enjoys learning new things and rarely updates his social media. John resides with his extremely patient and loving wife near Dayton, Ohio.

Jordan Borean

Role: Technical Editor

Jordan is a Software Engineer at [Red Hat](https://www.redhat.com/en)²³, working on the Windows integrations for Ansible. He originally worked on Java-based programs for a large company but felt the draw to open source software and has been an avid contributor since. Jordan mostly focuses on Python and PowerShell-based languages, and he is committed to trying to bridge the Windows and Linux worlds and make it easier for them to work with each other. Some projects that he works on are [pypsrp](https://github.com/jborean93/pypsrp)²⁴, [smbprotocol](https://github.com/jborean93/smbprotocol)²⁵, [pypsexec](https://github.com/jborean93/pypsexec)²⁶, and more recently [pyspnego](https://github.com/jborean93/pyspnego)²⁷. When finding some free time, Jordan blogs on [Blogging for Logging](https://www.bloggningforlogging.com/)²⁸ that cover technologies like PowerShell, Ansible, Windows protocols, and anything else that takes his fancy. You can usually get in contact with him on the [PowerShell Discord server](https://aka.ms/psdiscord)²⁹, or various IRC Freenode channels like #ansible, #Powershell, #packer-tool, and others.

¹⁸<https://twitter.com/ATXPowerShell>

¹⁹<https://twitter.com/AustinVMUG>

²⁰<https://twitter.com/vbrownbag>

²¹<https://twitter.com/jhoughes>

²²<https://www.fullstackgeek.net/>

²³<https://www.redhat.com/en>

²⁴<https://github.com/jborean93/pypsrp>

²⁵<https://github.com/jborean93/smbprotocol>

²⁶<https://github.com/jborean93/pypsexec>

²⁷<https://github.com/jborean93/pyspnego>

²⁸<https://www.bloggningforlogging.com/>

²⁹<https://aka.ms/psdiscord>

Kevin Laux

Role: Author and Quality Assurance Editor

Kevin is a manager for an orchestration platform team. He is passionate about PowerShell and has been leading training classes for his colleagues since the release of PowerShell v3. Kevin also serves as co-leader of the [Research Triangle PowerShell User Group](#)³⁰. In addition to PowerShell, he is always tinkering with new technology in his home lab and trying to learn everything he can. You can follow him on Twitter, [@rsrychro](#)³¹, and [GitHub](#)³².

Kieran Jacobsen

Role: Author and Technical Editor

Kieran Jacobsen (he/him) combines his passion for business process automation, systems integration, and cybersecurity to help organizations rapidly grow and evolve. Kieran's involvement in the technology community has seen him present at Microsoft's Ignite the Tour, NDC Sydney, and CrikeyCon. Kieran is well known for his security-focused presentations that blend real-world examples with storytelling. Microsoft has recognized Kieran's contributions to the community by awarding him with the Most Valuable Professional since 2017. Kieran is also a member of the GitKraken Ambassador Program. Kieran lives in Melbourne, Australia, with his husband and Burmese cat. In his spare time, Kieran enjoys computer games, Dungeons & Dragons, board games, and Melbourne's amazing food culture.

Kirill Nikolaev

Role: Author and Technical Editor

Kirill has more than 15 years of experience in IT, with specialization in Windows Server infrastructure, virtualization, information security, and automation. He began using PowerShell almost immediately after its release in 2006 and has been ever since. He is currently Head of the Windows Administration Team at [Fozzy.com](#)³³, an honest hosting provider, where he continues to give back to the community by sharing automation solutions through their [GitHub account](#)³⁴. You can follow him on Twitter, [@exchange12rocks](#)³⁵, or subscribe to his technical [blog](#)³⁶.

³⁰<https://rtpsug.com>

³¹<https://twitter.com/rsrychro>

³²<https://github.com/KevinLaux>

³³<https://fozzy.com>

³⁴<https://github.com/FozzyHosting>

³⁵<https://twitter.com/exchange12rocks>

³⁶<https://exchange12rocks.org>

Martha Clancy

Role: Linguistic Editor

Martha came to PowerShell and DevOps by way of database administration and is passionate about using code and automation to help everyone do their jobs more easily. You can find Martha on Twitter, [@marclancy](https://twitter.com/marclancy)³⁷, or her [blog](https://marthaclancy.com)³⁸.

Matt Corr

Role: Author

Husband, father, passionate about automation and process improvement. Matt has over 20 years of IT industry experience and is currently working as a DevOps Solution Specialist for [MOQdigital](https://www.moqdigital.com.au)³⁹. He is very passionate about PowerShell and is the go-to person in his teams for anything script or automation-related. He has experience with many build and deployment tools, such as Azure DevOps, Octopus Deploy, TeamCity, Terraform, and PowerShell. You can find Matt on Twitter ([@mattcorr](https://twitter.com/mattcorr)⁴⁰), [LinkedIn](https://www.linkedin.com/in/mattcorr/)⁴¹ or his [blog](https://www.intrepidintegration.com)⁴².

Michael B. Smith

Role: Quality Assurance Editor

Michael is an IT professional with over 35 years of experience in IT. Michael began using PowerShell during the Exchange 2007 Server beta and has been deeply into scripting with PowerShell ever since. He is a 13-time recipient of the Microsoft MVP award in Exchange Server. He has written many articles about Exchange, Active Directory, PowerShell, Windows Server, and Azure topics; and is passionate about presenting/training as well. You can find Michael on Twitter [@EssentialExch](https://twitter.com/EssentialExch)⁴³, at his blog [The Essential Exchange](https://www.essential.exchange)⁴⁴, and in the Facebook group [Azure Support](https://www.facebook.com/groups/AzureSupport)⁴⁵.

³⁷<https://twitter.com/marclancy>

³⁸<https://marthaclancy.com>

³⁹<https://www.moqdigital.com.au>

⁴⁰<https://www.twitter.com/mattcorr>

⁴¹<https://www.linkedin.com/in/mattcorr/>

⁴²<https://www.intrepidintegration.com>

⁴³<https://twitter.com/essentialexch>

⁴⁴<https://www.essential.exchange>

⁴⁵<https://www.facebook.com/groups/AzureSupport>

Michael Lotter

Role: Author

Michael has nearly a decade of system and network administration experience between South Africa and the US, from small shops to large enterprises, with a focus on automation and cybersecurity. He currently works in the finance sector as a systems engineer, where he uses PowerShell to automate complex processes. In addition to automation through PowerShell, he enjoys leveraging Azure and Intune to reduce organizations' on-premises footprint while maintaining high availability.

Michael Zanatta

Role: Author and Editor-in-Chief

Michael is a Microsoft MVP (Cloud and Datacenter Management), PowerShell SME, Speaker, Advocate, and Streamer, contracting as a PowerShell Developer for the Australian Federal Government. Michael has contributed to the PowerShell Conference Book Volume 2 and Volume 3, first as an author and stand-in editor on Volume 2 and then as the Senior Editor on Volume 3. You can follow him on Twitter, [@PowerShellMich1](https://twitter.com/PowerShellMich1)⁴⁶, or [LinkedIn](https://www.linkedin.com/in/michael-zanatta-61670258/)⁴⁷. Michael is a co-founder of the [Brisbane Infrastructure DevOps User Group](https://www.meetup.com/Brisbane-PowerShell-User-Group)⁴⁸ [YouTube channel](https://www.youtube.com/channel/UCQfLvFYohCCm_gTPEUfaAbw)⁴⁹ and author of his Livestream on [Twitch](https://www.twitch.tv/PowerShellMichael)⁵⁰.

Nicholas Bissell

Role: Author and Senior Editor

Though Nicholas's formal background is in research chemistry, he has over ten years of programming experience. In his spare time, he is a freelance software developer and mentor. Nicholas has previously worked as a sound engineer and a game developer. He is passionate about automation, open-source software, and the PowerShell community. You can find him on [GitHub](https://github.com/TheFreeman193)⁵¹, [StackOverflow](https://stackoverflow.com/users/12959131/thefreeman193)⁵², or [Reddit](https://www.reddit.com/user/thefreeman193)⁵³.

⁴⁶<https://twitter.com/PowerShellMich1>

⁴⁷<https://www.linkedin.com/in/michael-zanatta-61670258/>

⁴⁸<https://www.meetup.com/Brisbane-PowerShell-User-Group>

⁴⁹https://www.youtube.com/channel/UCQfLvFYohCCm_gTPEUfaAbw

⁵⁰<https://www.twitch.tv/PowerShellMichael>

⁵¹<https://github.com/TheFreeman193>

⁵²<https://stackoverflow.com/users/12959131/thefreeman193>

⁵³<https://www.reddit.com/user/thefreeman193>

Rob Derickson

Role: Quality Assurance Editor

Rob is an IT professional with 19 years in the field. Since 2008, PowerShell and the PowerShell community have been instrumental in his career success. You can find him on the [PowerShell Discord server](#)⁵⁴ and tweeting nothing on Twitter [@RobDerickson](#)⁵⁵.

Steven Judd

Role: Senior Editor

Steven Judd is a 25+ year IT Pro and currently a [Windows Systems Engineer at Meta Platforms Inc.](#)⁵⁶ with an emphasis on Enterprise Messaging and Digital Loss Prevention. He has been using PowerShell since 2010. He was an author and editor on [The PowerShell Conference Book Volume 3](#)⁵⁷, has co-developed a custom training program for PowerShell, and is an occasional presenter at PowerShell user groups. He loves to help people learn and recognize the value of automation. He spends his free time learning more about PowerShell, digital security, and cloud technologies, along with creating and telling [Dad jokes](#)⁵⁸. You can find him hanging out on the [PowerShell Discord Server](#)⁵⁹ bridge channel, taking care of his family, running marathons, playing the cello, plus a handful of other hobbies he can't seem to quit. Please follow him on Twitter, [@stevenjudd](#)⁶⁰, read his [blog](#)⁶¹, and review, use, and improve his code on [GitHub](#)⁶².

Wes Stahler

Role: Technical Editor

Wes Stahler has over 25 years of Information Technology experience as a Developer, Systems Administrator, and Manager of an Identity and Access Management team. He enjoys evangelizing PowerShell's merits and has presented numerous times nationally at the Microsoft Health Users Group and locally for the Central Ohio PowerShell Users Group. He strives to automate within Exchange and Active Directory and advocates on the "Power of the Shell." Available on Twitter at [@stahler](#)⁶³.

⁵⁴<https://aka.ms/psdiscord>

⁵⁵<https://twitter.com/RobDerickson>

⁵⁶<https://www.linkedin.com/in/stevenjudd/>

⁵⁷<https://leanpub.com/psconfbook3>

⁵⁸<https://www.youtube.com/watch?v=BZZM6i8AE1Y>

⁵⁹<https://aka.ms/psdiscord>

⁶⁰<https://twitter.com/stevenjudd/>

⁶¹<https://blog.stevenjudd.com/>

⁶²<https://github.com/stevenjudd>

⁶³<https://twitter.com/stahler>

Acknowledgements

This book was made possible by a multitude of people, not just the initial team of editors and the writers, but their family, friends, mentors, peers, and—most of all—you, the readers.

By reading this book, you're helping to make sure that our field expands and grows, creating opportunities for folks who otherwise might never see them. That's incredible and needs no qualifiers.

This project owes itself to the PowerShell community and everyone who gave it their time, energy, and money.

Disclaimer

All code examples shown in this book have been tested by each chapter author and every effort has been made to ensure that they're error-free. However, since every environment is different, the examples should be run in a non-production environment and should be thoroughly tested before being used in a production environment. It's recommended that you use a non-production or lab environment to thoroughly test code examples used throughout this book.

All data and information provided in this book is for educational purposes only. The editors make no representations as to the accuracy, completeness, currentness, suitability, or validity of any information in this book and won't be liable for any errors, omissions, or delays in this information or any losses, injuries, or damages arising from its display or use. All information is provided on an as-is basis.

This disclaimer is provided simply because someone, somewhere will ignore this disclaimer and if they do experience problems or a "resume generating event," they have no one to blame but themselves. Don't be that person!

Introduction

By Michael Zanatta

Hello Reader!

Modern Automation with PowerShell was an initiative between Steven Judd and myself to create a textbook as a love-letter *for the community by the community*. We wanted to provide an intermediary resource, different in style from the previous PowerShell Conference Books. We wanted to focus on a deeper understanding of the inner workings of PowerShell, share best practices and tips, and have this book serve as a study resource and lesson guide. All royalties for this book will go to the “OnRamp” program (see below).

To our amazing Authors and Editors, and also their Partners and Families, thank you for your time and sacrifice. This book could not exist without you.

To you, the Reader, I hope you enjoy reading this book as much as we enjoyed writing it.



This is a Leanpub “Agile-published” book. All the work for this book has been completed. However, as issues are reported, supplementary updates may be released. Leanpub will send out an email when the book is updated. These revisions will be available at no extra charge. To provide feedback, use the “Email the Authors” link on [the book’s Leanpub web page](#)⁶⁴. Whether it’s a code error, a typo, or a request for clarification, our editors will review your feedback, make changes, and re-publish the book. Unlike the traditional paper publishing process, your feedback can have an immediate effect.

About OnRamp

OnRamp is an entry-level education program focused on PowerShell and Development Operations. It is a series of presentations that are held at the [PowerShell + DevOps Global Summit](#)⁶⁵ and is designed for entry-level technology professionals who have completed foundational certifications such as CompTIA A+ and Cisco IT Essentials. No prior PowerShell experience is required. Basic knowledge of server administration is beneficial. OnRamp ticket holders will be able to network with other Summit attendees who are attending the scheduled Summit sessions during keynotes, meals, and evening events.

Through fundraising and corporate sponsorships, [The DevOps Collective, Inc.](#)⁶⁶ will be offering several full-ride scholarships to the OnRamp track at the PowerShell + DevOps Global Summit.

All (100%) of the royalties from this book are donated to the OnRamp scholarship program.

⁶⁴<https://leanpub.com/modernautomationwithpowershell>

⁶⁵<https://powershell.org/summit/>

⁶⁶<https://devopscollective.org/>

More information about the [OnRamp track](#)⁶⁷ at the PowerShell + DevOps Global Summit and [their scholarship program](#)⁶⁸ can be found on the [PowerShell.org](#)⁶⁹ website.

See the [DevOps Collective Scholarships cause](#)⁷⁰ on [Leanpub.com](#)⁷¹ for more books that support the OnRamp scholarship program.

Prerequisites

Prior experience with PowerShell is recommended. This book is written for intermediate audiences with intermediate experience with PowerShell.

A Note on Code Listings



Paperback Readers can access digital copies of examples from the book at:
<https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/tree/main/Edition-01/Examples>⁷²

If you've read other PowerShell books from LeanPub, you probably have seen some variation on this code sample disclaimer. The code formatting in this book only allows for about 75 characters per line before the text will start automatically wrapping. All attempts have been made to keep the code samples within that limit, although sometimes you may see some awkward formatting as a result.

For example:

```
Get-CimInstance -ComputerName $computer -ClassName Win32_LogicalDisk -Filter 'DriveType\
e=3' -Property DeviceID, Size, FreeSpace
```

Here, you can see the default action for a line that is too long—it gets word-wrapped, and a backslash is inserted at the wrap point to let you know it has wrapped. Attempts have been made to avoid those situations, but they may sometimes be unavoidable. Instead of having a long command that wraps, [splatting](#)⁷³ is used instead.

⁶⁷<https://powershell.org/summit/summit-onramp/>

⁶⁸<https://powershell.org/summit/summit-onramp/onramp-scholarship/>

⁶⁹<https://powershell.org/>

⁷⁰<https://leanpub.com/causes/devopscollective>

⁷¹<https://leanpub.com/>

⁷²<https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/tree/main/Edition-01/Examples>

⁷³https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_splatting?view=powershell-7

```
$params = @{  
    ComputerName = $computer  
    ClassName    = 'Win32_LogicalDisk'  
    Filter       = 'DriveType=3'  
    Property     = 'DeviceID', 'Size', 'FreeSpace'  
}  
  
Get-CimInstance @params
```



If you read this book on a Kindle, tablet, or another e-reader, use the PDF manuscript, not EPUB. EPUB has known formatting issues with Block Types (For Example, Warning, Tips, Errors), Code Samples, and Annotations.

When writing PowerShell expressions, you shouldn't be limited by these constraints. All downloadable code samples will be in their original form.

Feedback

Have a question, comment, or feedback about this book? Please share it via the Leanpub forum dedicated to this book. Once you've purchased this book, log in to Leanpub, and click on the "Join the Forum" link in the Feedback section of [this book's webpage](https://leanpub.com/modernautomationwithpowershell)⁷⁴.

⁷⁴<https://leanpub.com/modernautomationwithpowershell>

Using Regexes

“Garbage in, Garbage out.” — George Fuechsel, Raymond Crowley, et al.

Administrators must often work with external data, so pattern matching is a valuable tool. Regex patterns play an important role in transforming and bringing data into the PowerShell environment. This role becomes even more significant now that PowerShell is cross-platform. This section takes you right from the beginning, through to advanced uses for regexes, including:

- A 101-style introduction to regexes.
- Accessing and using regex patterns in PowerShell.
- A deep dive into PowerShell and .NET regex.
- Best practices for using regexes in PowerShell.

Regex 101

Regular expressions, or **regexes**⁷⁵, are a powerful pattern-matching technology and language for parsing plain text into usable objects. In PowerShell, much of the data you work with is already in this form. A regex is therefore most useful when processing the output from external applications or environments, such as log files and templates. You may have happened upon regexes in other learning materials, or other programming languages and environments. They might have appeared to be a daunting subject or otherwise a niche area of expertise. Developers, engineers, and administrators use these patterns often, if not daily. They're an essential tool in your toolkit and can become intuitive with a basic understanding of how regex engines process text strings.



Did you know?

Mathematician Stephen Cole Kleene developed *regular language*, the concept behind regular expressions, in the early 1950s.⁷⁶ It was almost twenty more years before the first computer implementations existed. An early adopter of the syntax was Ken Thompson's Quick Editor (QED) in 1968.⁷⁷

This chapter recaps the fundamental syntax and structure of regex patterns and explains how to use these in the PowerShell environment. [Accessing Regexes](#) introduces more complex constructs, solving relevant examples to explain these and demonstrate the power of regexes. [Regex Deep Dive](#) discusses deconstructing and debugging your patterns, looking from the perspective of the regex engine, and getting familiar with the machinery within. It also covers the remaining syntax to complete your PowerShell regex toolkit. Finally, [Regex Best Practices](#) rounds off with some best practices, design strategies, and where to go for more on regexes in PowerShell and in general.

First Principles and Limitations

There is some debate about whether modern regexes can still be considered regular expressions. The regex chapters use the terms **regex** and **regexes** to avoid confusion.

Regexes are, in essence, instructions that tell a regex engine how to read some text for you. They define patterns to *match* in a text string and can *capture* substrings. You can use these *captures*

⁷⁵Though **regexes** is the correct plural, the uncountable plural form **regex** is also common when referring to the topic as a whole, as is **regex** to describe the underlying theory.

⁷⁶S. C. Kleene. (1956). *Representation of Events in Nerve Nets and Finite Automata*. In: Automata Studies, (AM-34), pp. 3–42. C. E. Shannon and J. McCarthy (eds.). Princeton University Press. DOI: [10.1515/9781400882618-002](https://doi.org/10.1515/9781400882618-002).

⁷⁷K. Thompson. (1968). *Programming Techniques: Regular expression search algorithm*. Commun. ACM, vol. 11, no. 6, pp. 419–422. DOI: [10.1145/363347.363387](https://doi.org/10.1145/363347.363387).

as *backreferences* later in the pattern, and they're also returned by the engine back into the programming environment. Captures are also used to substitute a replacement into the input string.

Regexes have limitations, of course. The idea of *Garbage In, Garbage Out* (GIGO) is relevant here. The regex engine interprets the provided pattern sequentially, reading through the input text and backtracking as necessary until no more matches are possible. It can't observe the input text as a whole or make decisions from a *big picture* perspective, as humans can. It's therefore prudent to make use of the programming environment to supplement your regex patterns. You can find out more about this later.



PowerShell regex uses the *.NET* regex engine. Examples in this book can be applied to other *.NET* languages (C#, VB.NET, F#, and ASP.NET).

Wildcard Patterns vs. Regexes

PowerShell also supports **wildcard expressions**. These are much simpler pattern matching expressions that don't support capturing. You can use these with the `-like` and `-notlike` operators, and with cmdlet parameters that support wildcards. The [Advanced Conditions](#) chapter covers wildcard patterns comprehensively.

Differences Between PowerShell Regexes and Others

PowerShell's regex flavor is largely based on Perl 5⁷⁸. The flavor hasn't changed since 2.0 and is therefore relevant to all supported editions of PowerShell at the time of writing.

There are several differences between PowerShell regexes and other common flavors, such as Perl-Compatible Regular Expressions (PCRE), Python, and Java.⁷⁹ The more important ones are:

Not Supported:

- Pattern recursion (alternative: *balancing groups*)
- Possessive quantifiers (alternative: *atomic groups*)
- Unicode scripts (alternative: *Unicode blocks*)
- Unicode whole graphemes (alternative: individual code points)
- Global `/g` modifier (alternative: use `[regex]::Matches()` instead of `::Match()` or `-match`)
- Subroutines
- Branch reset groups
- Literal text spans
- Character class intersection

Supported:

⁷⁸Microsoft. (2020, Jun. 30). *.NET regular expressions*. Microsoft Docs. [Online]. Available: <https://learn.microsoft.com/en-us/dotnet/standard/base-types/regular-expressions>. [Accessed: Jun. 12, 2021].

⁷⁹J. E. F. Friedl. (2006). *Mastering Regular Expressions*. 3rd ed. Beijing [u.a.]: O'Reilly. ISBN: 978-0-596-52812-6.

- Named groups (?<name>) or (? 'name')
- Backreferences \NN and \k<name> or \k 'name'
- Conditional subexpressions
- Atomic groups
- Unicode
 - Code points \uHHHH
 - Blocks \p{Is<BlockName>}
 - Categories Short form/key: \p{X} or \p{Xy}
- Balancing groups (?<First>...)(?<Second-First>...)
- Inline comments (?# Comment)
- Inline options (?msnix-msnix)
- Option spans (?msnix-msnix:...)
- Variable-length lookarounds, including lookbehinds
 - Positive lookahead (?=...)
 - Negative lookahead (?!...)
 - Positive lookbehind (?<=...)
 - Negative lookbehind (?<!=...)

This list is primarily for reference, and you can learn more about the topics described later.

Getting Started



This chapter covers the basics of regexes. If you already have an intermediate understanding of regexes, you can consider skipping ahead to the [Accessing Regexes](#) chapter.

However, there are lots of useful pointers, so reading through will help set you up for later.

To introduce the fundamentals of regexes, consider this example. The days of the week in the English language all end in ‘day’, with three to six preceding letters. You could simply match each day individually.

Example 1: Matching 'Friday'

```
1 $MyString = 'Fridays are the best days.'
2
3 $MyString -match 'Monday'
4 $MyString -match 'Tuesday'
5 $MyString -match 'Wednesday'
6 $MyString -match 'Thursday'
7 $MyString -match 'Friday'
8 $MyString -match 'Saturday'
9 $MyString -match 'Sunday'
```

```
False
False
False
False
True
False
False
```

This works because the regex engine interprets the letters of *Friday* and *Saturday* literally. To avoid using the `-match` operator many times, you can use **alternation**. Think of this as a *logical OR*. Regexes use the pipe `|` character for alternation.

Example 2: Matching any day of the week

```
1 $MyString = 'It rained on Friday, but Monday will be clear.'
2
3 $MyPattern = 'Monday|Tuesday|Wednesday|Thursday|Friday|Saturday|Sunday'
4
5 $MyString -match $MyPattern
```

```
True
```

Character Classes

What about matching any word ending in 'day'? **Character classes** match one character from a **list**. Perhaps the most common of these is `\w`, the *word character* class. This matches letters, numbers, and marks that aren't punctuation (except the underscore `_`) and aren't white space. PowerShell regexes support Unicode, so this includes word characters for many other languages, too. In fact, of the first 65,536 Unicode code points, `\w` matches 50,320 of them. For everyday use with English language characters, this matches A–Z, a–z, 0–9, and `_`.

Example 3: Matching 4 letters before ‘day’

```

1 $MyRegex = '\w\w\w\wday'
2
3 'Tomorrow is a holiday.' -match $MyRegex
4 'Tomorrow is Monday.' -match $MyRegex
5 'Tomorrow is Wednesday.' -match $MyRegex

```

```

True
False
True

```

Notice the backslash `\` used with the `w` here. Backslashes are **metacharacters** in regexes—they have a special meaning. They denote special tokens, including character classes. They can also denote escape sequences.

1. *holiday* results in a match, since `\w\w\w\w` matches ‘**holi**’.
2. *Monday* results in no match, because `\w` doesn’t match spaces, and there are only three letters before ‘**day**’.
3. *Wednesday* results in a match, even though there are more than four letters before ‘**day**’. This is because the pattern can still match `\w\w\w\w` with ‘**dnes**’. **Anchors** overcome this problem. You can read more about them later in the chapter.

There are three other predefined character classes in PowerShell regexes:

- To match **numeric digits**, use `\d` (for everyday English, think 0–9).
- To match **space characters**, use `\s` (for everyday English, think spaces, tabs, and newlines).
- To match **any character** except a newline, use a period (`.`).

There are also three **inverse** character classes matching any character that **isn’t** in that class. Each uses the same letter but capitalized: `\W \D \S`.

Example 4: Using character classes and their inverses

```

1 'ab' -match '\S\s'
2 'ab' -match '\S\S'
3 'ab c1' -match '\w\w\s\D\d'

```

```

False
True
True

```

The first pattern is looking for one not-space and one space, but the string contains no spaces. The second pattern is looking for two not-spaces, which match ‘**ab**’. The third pattern is looking for two word characters, a space, one not-decimal, and one decimal. This matches ‘**ab c1**’, as ‘**c**’ isn’t a decimal.



There are also escape sequences that match single characters, such as a newline. You can find out about these later in the chapter.

Custom Character Classes

If you need to define your own set of characters to match, use square brackets [...]. This can be useful for narrowing the scope when searching for specific text, such as punctuation or disallowed characters in user input.

Example 5: Using a custom class to match punctuation

```
1 $MyPattern = '\w[.,'':;?!]'
2
3 'This sentence has no punctuation' -match $MyPattern
4 'This is punctuated with a period.' -match $MyPattern
5 'The period in "1.2" also matches' -match $MyPattern
```

```
False
True
True
```

Note the two single quotation marks `' '`, which insert a single mark `'` into a literal string. Observe also that a period `.` inside a custom character class matches only a **period**, not *any character*.

Custom character classes are also capable of inversion. By inserting a caret `^` after the opening square bracket `[`, the class matches **any character** not in it.



The following example uses the `$Matches` automatic variable. The next chapter, [Accessing Regexes](#), discusses this further. For now, know that PowerShell sets `$Matches` each time you call `-match` and it succeeds. The entry `$Matches[0]` reveals what the **entire pattern** matched in the input string.

Example 6: Inverse character classes

```
1 $MyPattern = '"[^"]+"'
2
3 'This is a string with a "quoted" word' -match $MyPattern
4 $Matches[0]
```

```
True
"quoted"
```

[Example 6](#) matches one or more characters that aren't quotation marks, surrounded by quotation marks. The plus `+` is a **quantifier** meaning, *match **one or more** instances of the last token*.



Character Classes Reference

You can view a complete reference for character classes at [Microsoft Docs](#)⁸⁰.

⁸⁰<https://learn.microsoft.com/en-us/dotnet/standard/base-types/character-classes-in-regular-expressions>

Quantifiers

Repeating literal characters or character classes for large match spans would take up a lot of space. Regexes make use of **quantifiers** to control the number of allowed repetitions of the preceding token in the pattern. You can rewrite the pattern from earlier, `\w\w\w\wday`, as `\w{4}day`. What about a range of repeats?

Example 7: Using the range quantifier

```
1 $MyPattern = '\w{3,6}day'
2
3 'Today' -match $MyPattern # 2 letters before 'day'
4 'Tuesday' -match $MyPattern # 4 letters before 'day'
```

False
True

This results in the general formula `{min,max}`. Don't use a space character on either side of the comma. You can omit max, leaving the comma `(,)` in place, to match *min or more*.

Example 8: Using the 'at least' quantifier

```
1 # 6 or more word characters before 'day'
2 $MyPattern = '\w{6,}day'
3
4 'Tuesday' -match $MyPattern # 4 letters before 'day'
5 'Wednesday' -match $MyPattern # 6 letters before 'day'
6 'Postholiday' -match $MyPattern # 8 letters before 'day'
```

False
True
True

There are also shorthand quantifiers for common requirements.

- Plus `+`, which matches **one or more** repetitions (think `{1,}`)
- Star `*`, which matches **zero or more** repetitions (think `{0,}`)
- Question mark `?`, which matches **zero or one** repetition (think `{0,1}`)

You can use the **zero or one** quantifier to make tokens optional.

Example 9: Using the ‘zero or one’ quantifier

```

1 $MyPattern = 'h?ello'
2
3 'hello' -match $MyPattern
4 'yellow' -match $MyPattern

```

```

True
True

```

Example 10: The ‘zero or more’ quantifier, and greedy matching

```

1 $MyPattern = 'ab.*ly'
2
3 'absolutely' -match $MyPattern
4 $Matches[0]
5
6 'ably' -match $MyPattern
7 $Matches[0]
8
9 'absolutely lovely' -match $MyPattern
10 $Matches[0]

```

```

True
absolutely

```

```

True
ably

```

```

True
absolutely lovely

```

The first match understandably succeeds, with `.*` matching ‘**solute**’. The second proves that `*` can match zero times. What’s going on with number three, then? Wouldn’t it just match ‘**absolutely**’?

Going back to the idea of **sequential** processing, the engine performs the following steps:

1. The regex engine matches `a` to ‘**a**’, then `b` to ‘**b**’, so it moves on.
2. `.*` matches zero **or more** characters, and the engine doesn’t yet care what comes next in the pattern. Therefore, it matches ‘**solutely lovely**’.
3. The engine tries to match `l`, but there are no more characters, so the `.*` gives up the last ‘**y**’.
4. The engine still can’t match `l` to ‘**y**’, so the `.*` gives up the last ‘**l**’.
5. The engine now matches `l` to ‘**l**’ and moves on.
6. Finally, the engine matches `y` to ‘**y**’, and has now matched the pattern fully, so processing stops.

Matching **as much as possible**, then giving back as needed, is **greedy** matching. This is the default in PowerShell regexes. The opposite of greedy is **lazy** matching, where the quantifier matches **as little as possible** and takes more if needed. You can make any quantifier lazy by adding a question mark `?` after it:

- `+?`: Matches one or more lazily
- `*?`: Matches zero or more lazily
- `??`: Matches zero or one lazily
- `{min,max}?`: Matches between *min* and *max* times lazily
- `{min,}?:` Matches *min* or more lazily

With this in mind, the lazy `.*` fixes the last match in [Example 10](#).

Example 11: Using the ‘lazy quantifier’ modifier

```
1 $MyPattern = 'ab.*?ly'
2
3 'absolutely lovely' -match $MyPattern
4 $Matches[0]
```

```
True
absolutely
```

The engine only matches ‘**absolutely**’ this time. Initially, `.*` only matches the first ‘s’ but takes more characters until the next token `l` matches the first ‘l’. This matches, but the following ‘u’ doesn’t. Therefore, the engine **backtracks** in the pattern, with `.*` taking more characters until the next token `l` matches the second ‘l’. The final token `y` then matches the last ‘y’ and processing stops with a successful match.

The [Regex Deep Dive](#) chapter discusses [backtracking and branching](#) in more detail.



Quantifiers Reference

You can view a complete reference for quantifiers at [Microsoft Docs](#)⁸¹.

Character Escape Sequences

Character escape sequences match a single character instead of a character from a range, like character classes. You can use them to match characters you can’t type, or would otherwise have a special meaning (think `?`, `*`, or `\`, for example).



The following example uses a literal here-string. Use these to create multiline strings in PowerShell. Expandable here-strings `@"` and `"@"` also exist.

⁸¹<https://learn.microsoft.com/en-us/dotnet/standard/base-types/quantifiers-in-regular-expressions>

Example 12: Matching line endings with character escape sequences

```

1 $MyPattern = '\r?\n'
2
3 '@'
4 This is a uniline (single line) string
5 '@ -match $MyPattern
6
7 '@'
8 This is
9 a multiline string
10 '@ -match $MyPattern

```

False

True

Example 12 matches line endings for both Windows and Unix-like systems. This is useful when working in PowerShell 6.0 and later, which is cross-platform. The first part of the pattern, `\r?`, matches an optional single *carriage return* (0x0D). The second part, `\n`, matches a *line feed* (0x0A).

You can escape character classes too. Escaping the backslash `\` causes the engine to interpret it literally. The sequence `\\n` matches a backslash followed by an ‘n’, not a line feed.



Character Escape Sequences Reference

You can view a complete reference for escape sequences at [Microsoft Docs](https://learn.microsoft.com/en-us/dotnet/standard/base-types/character-escapes-in-regular-expressions)⁸².

Anchors (*Zero-Width Assertions*)

The pattern `\w{3,6}day` from **Example 7** matches all days of the week, but also longer words. The engine matches the word ‘**post**holiday’ after backtracking twice, skipping the ‘**po**’. Is this preventable? The answer is yes, with **anchors**. These are assertions that set conditions for matches but don’t take any characters from the string. Therefore, they’re also known as *zero-width* assertions.

Perhaps the most common of these is the **caret** `^` and **dollar** `$`. A caret matches the **beginning** of the string, so `^aa\w+` matches ‘aardvark’, but not ‘the aardvark’. A dollar matches the **end** of the string, so `ero$` matches ‘zero’, but not ‘zeroes’. *Multiline mode*, a regex option, causes these anchors to match the beginning and end of lines, too. You can read more about [regex options](#) in the [Regex Deep Dive](#) chapter.

Another commonly used anchor is the **word boundary** `\b`. This matches the boundary between a word `\w` character and a not-word `\W` character. It also has an inverse, the not-word-boundary `\B`. This can never match a word boundary.

⁸²<https://learn.microsoft.com/en-us/dotnet/standard/base-types/character-escapes-in-regular-expressions>

Another kind of zero-width assertion is the **lookaround**. This feature matches one or more regex tokens in a subexpression while still being zero-width. The [Regex Deep Dive](#) chapter discusses [subexpressions](#) further.



Anchors Reference

You can view a complete reference for regex anchors at [Microsoft Docs](#)⁸³

Captures

The final topic that this chapter covers is the **capturing subexpression**, or *capturing group*. These enable you to extract substrings from the input string and use brackets (parentheses) (and). You can wrap any part of a regex pattern to make it a capturing subexpression. The engine assigns it a number starting from 1, and if it makes a match, it returns the *captured* substring into the programming environment. The next example extracts the original word from an *infix*ed one.



The terms *capturing group* and *capturing subexpression* are often used interchangeably. Technically, a subexpression is any part of a regex pattern delimited by grouping constructs. Capturing subexpressions create capturing groups, which capture matches made by the subexpressions within. To avoid confusion, this chapter uses the term *group* for both the construct and the resulting group.

Example 13: Extracting infix text with a capturing group

```
1 $MyPattern = '\b(\w+)-\w{6,}-(\w+)\b'
2
3 'This is fan-flaming-tastic!' -match $MyPattern
4 $Matches[0]
5
6 $Matches[1], $Matches[2] -join ''
```

```
True
fan-flaming-tastic

fantastic
```

This pattern may seem a bit complicated but, when broken down, it's straightforward.

- The word boundary `\b` anchors at each end force the engine to match no more or less than the infix word.
- The first `(\w+)` matches **and captures** the part of the word before the first hyphen.
- The `-\w{6,}-` matches an infixation with at least six letters. This eliminates most other multi-word phrases like *'up-to-date'*, but not phrases with longer middle words, like *'well-thought-out'*.

⁸³<https://learn.microsoft.com/en-us/dotnet/standard/base-types/anchors-in-regular-expressions>

- The last `(\w+)` matches **and captures** the part of the word after the last hyphen.



Try the examples in this chapter in PowerShell and observe how changing the input or pattern changes the output.

Experimentation is a great way to familiarize yourself with regexes.

The `$Matches` automatic variable stores these numbered captures as entries in a hashtable. Recall that the zeroth entry `$Matches[0]` is the *capture* of the entire pattern. It's also possible to group regex tokens without capturing them. This is a **non-capturing group** and takes the form `(?:...)` where `...` is any regex subexpression.



The `$Matches` automatic variable isn't nullified before each regex operation. This means that if a match fails, `$Matches` will still contain the result of the last successful match.⁸⁴ You should therefore check for match success before reading this variable.

Visualizing Captures

Sometimes, it can be a little hard to visualize what matches, groups, and captures actually are. The following example provides a helpful representation using .NET methods. You can find out more about the .NET methods in the [Accessing Regexes](#) chapter.

Example 14: Visualizing captures

```

1 $Result = [regex]::Matches(
2     'abcdefg hijklmn', '(?:(<first>\w)(?<second>\w))+')
3 )
4
5 if ($Result.Success) {
6     $matchCount = -1
7     $Result.ForEach{
8         Write-Host ('Match {0}:' -f ++$matchCount)
9         $groupCount = -1
10        $_.Groups.ForEach{
11            Write-Host ('  Group {0} (Name = {1}):' -f ++$groupCount, $_.Name)
12            $captureCount = -1
13            $_.Captures.ForEach{
14                Write-Host (
15                    '    Capture {0} (pos {1}) = {2}' -f
16                    ++$captureCount, $_.Index, $_.Value
17                )
18            }
19        }
20    }
21 } else { Write-Host 'No matches' }
```

⁸⁴Microsoft. (2021, Oct. 27). *About Automatic Variables (Microsoft.PowerShell.Core) - Matches*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_automatic_variables#matches. [Accessed: Nov. 15, 2021].

```

Match 0:
  Group 0 (Name = 0):          <-- Entire match 'abcdef'
    Capture 0 (pos 0) = abcdef <-- $0 or $& in replacement pattern
  Group 1 (Name = first):      <-- From subexpression (?<first>\w)
    Capture 0 (pos 0) = a
    Capture 1 (pos 2) = c
    Capture 2 (pos 4) = e      <-- $1 or ${first} in replacement pattern
  Group 2 (Name = second):     <-- From subexpression (?<second>\w)
    Capture 0 (pos 1) = b
    Capture 1 (pos 3) = d
    Capture 2 (pos 5) = f      <-- $2 or ${second} in replacement pattern

Match 1:
  Group 0 (Name = 0):          <-- Entire match 'hijklm'
    Capture 0 (pos 8) = hijklm <-- $0 or $& in replacement pattern
  Group 1 (Name = first):      <-- (From subexpression (?<first>\w)
    Capture 0 (pos 8) = h
    Capture 1 (pos 10) = j
    Capture 2 (pos 12) = l     <-- $1 or ${first} in replacement pattern
  Group 2 (Name = second):     <-- From subexpression (?<second>\w)
    Capture 0 (pos 9) = i
    Capture 1 (pos 11) = k
    Capture 2 (pos 13) = m     <-- $2 or ${second} in replacement pattern

```



Grouping Constructs Reference

You can view a complete reference for grouping constructs at [Microsoft Docs](https://learn.microsoft.com/en-us/dotnet/standard/base-types/grouping-constructs-in-regular-expressions)⁸⁵.

⁸⁵<https://learn.microsoft.com/en-us/dotnet/standard/base-types/grouping-constructs-in-regular-expressions>