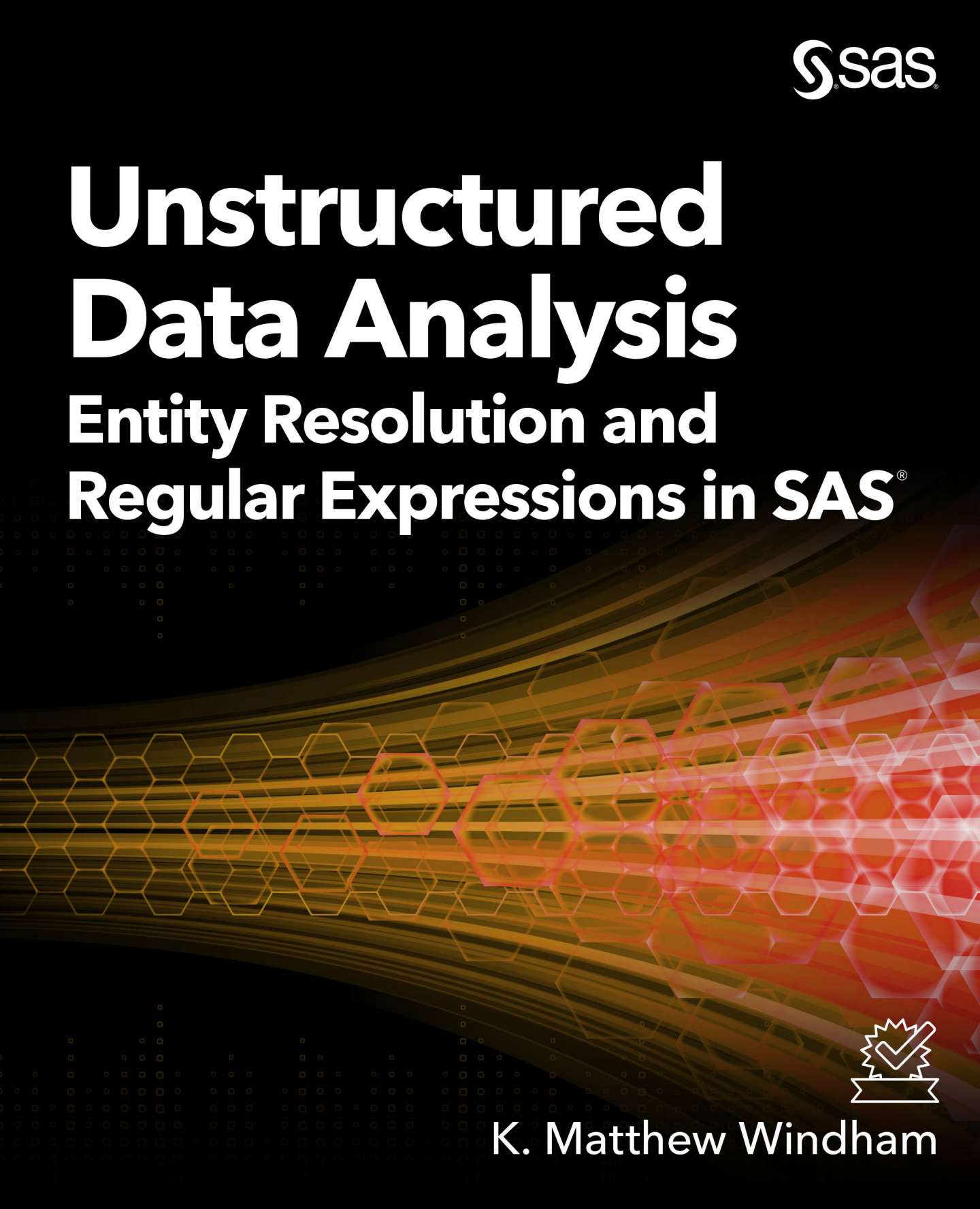
sas
Unstructured Data Analysis
Entity Resolution and Regular Expressions in SAS®
K. Matthew Windham

The correct bibliographic citation for this manual is as follows: Windham, Matthew. 2018. *Unstructured Data Analysis: Entity Resolution and Regular Expressions in SAS®*. Cary, NC: SAS Institute Inc.

Unstructured Data Analysis: Entity Resolution and Regular Expressions in SAS®

978-1-62959-842-0 (Hardcopy)
978-1-63526-711-2 (Web PDF)
978-1-63526-709-9 (epub)
978-1-63526-710-5 (mobi)

SAS Institute Inc., SAS Campus Drive, Cary, NC 27513-2414

September 2018

Contents

About This Book

What Does This Book Cover?

This book was written to provide readers with an introduction to the vast world that is unstructured data analysis. I wanted to ensure that SAS programmers of many different levels could approach the subject matter here, and come away with a robust set of tools to enable sophisticated analysis in the future.

I focus on the regular expression functionality that is available in SAS, and on presenting some basic data manipulation tools with the capabilities that SAS has to offer. I also spend significant time developing capabilities the reader can apply to the subject of entity resolution from end to end.

This book does not cover enterprise tools available from SAS that make some of the topics discussed herein much easier to use or more efficient. The goal here is to educate programmers, and help them understand the methods available to tackle these things for problems of reasonable scale. And for this reason, I don't tackle things like entity resolution in a "big data" context. It's just too much to do in one book, and that would not be a good place for a beginner or intermediate programmer to start.

Performing an array of unstructured data analysis techniques, culminating in the development of an entity resolution analytics framework with SAS code, is the central focus of this book. Therefore, I have generally arranged the chapters around that process. There is foundational information that must be covered in order to enable some of the later activities. So, Chapters 1 and 2 provide information that is critical for Chapter 3, and that is very useful for later chapters.

Chapter 1: Getting Started with Regular Expressions

In order to effectively prepare you for doing advanced unstructured data analysis, you need the fundamental tools to tackle that with SAS code. So, in this chapter, I introduce regular expressions.

Chapter 2: Using Regular Expressions in SAS

In this chapter, I will begin using regular expressions via SAS code by introducing the SAS functions and call routines that allow us to accomplish fairly sophisticated tasks. And I wrap up the chapter with some practical examples that should help you tackle real-world unstructured data analysis problems.

Chapter 3: Entity Resolution Analytics

I will introduce entity resolution analytics as a framework for applying what was learned in chapters 1 and 2 in combination with techniques introduced in the subsequent chapters of this book. This framework will be guiding force through the remaining chapters of this book, providing you with an approach to begin tackling entity resolution in your environment.

Chapter 4: Entity Extraction

Leveraging the foundation established in Chapters 1 and 2, I will discuss methods for extracting entity references from unstructured data sources. This should be a natural extension of the work that was done in Chapter 2, with a particular focus—preparing for the entity resolution.

Chapter 5: Extract, Transform, Load

I will cover some key ETL elements needed for effective data preparation of entity references, and demonstrate how they can be used with SAS code.

Chapter 6: Entity Resolution

In this chapter, I will walk you through the process of actually resolving entities, and acquaint you with some of the challenges of that process. I will again have examples in SAS code.

Chapter 7: Entity Network Mapping and Analysis

This chapter is focused on the steps taken to construct entity networks and analyze them. After the entity networks have been defined, I will walk through a variety of analyses that might be performed at this point (this is not an exhaustive list).

Chapter 8: Entity Management

In this chapter, I will discuss the challenges and best practices for managing entities effectively. I try to keep these guidelines general enough to fit within whatever management process your organization uses.

Appendix A: Additional Resources

I have included a few sections for random entity generation, regular expression references, Perl version notes, and binary/hexadecimal/ASCII code cross-references. I hope they prove useful references even after you have mastered the material.

Is This Book for You?

I wrote this book for ambitious SAS programmers who have practical problems to solve in their day-to-day tasks. I hope that it provides enough introductory information to get you started, motivational examples to keep you excited about these topics, and sufficient reference material to keep you referring back to it.

To make the best use of this book, you should have a solid understanding of Base SAS programming principles like the DATA step. While it is not required, exposure to PROC SQL and macros will be helpful in following some of the later code examples.

This book has been created with a fairly wide audience in mind—students, new SAS programmers, experienced analytics professionals, and expert data scientists. Therefore, I have provided information about both the business and technical aspects of performing unstructured data analysis throughout the book. Even if you are not a very experienced analytics professional, I expect you will gain an understanding of the business process and implications of unstructured data analysis techniques.

At a minimum, I want everyone reading this book to walk away with the following:

- A sound understanding of what both regular expressions and entity resolution are (and aren't)
- An appreciation for the real-world challenges involved in executing complex unstructured data analysis
- The ability to implement (or manage an implementation) of the entity resolution analytics methodology discussed later in this book

- An understanding of how to leverage SAS software to perform unstructured data analysis for their desired applications

The SAS Platform is quite broad in scope and therefore provides professionals and organizations many different ways to execute the techniques that we will cover in this book. As such, I can't hope to cover every conceivable path or platform configuration to meet an organization's needs. Each situation is just different enough that the SAS software required to meet that organization's scale, user skill level(s), financial parameters, and business goals will vary greatly.

Therefore, I am presenting an approach to the subject matter which enables individuals and organizations to get started with the unstructured data analysis topics of regular expressions and entity resolution. The code and concepts developed in this book can be applied with solutions such as SAS Viya to yield an incredible level of flexibility and scale. But I am limiting the goals to those that can yield achievable results on a small scale in order for the process and techniques to be well understood. Also, the process for implementation is general enough to be applied to virtually any scale of project. And it is my sincere hope that this book provides you with the foundational knowledge to pursue unstructured data analysis projects well beyond my humble aim

What Should You Know about the Examples?

This book includes tutorials for you to follow to gain hands-on experience with SAS.

Software Used to Develop the Book's Content

SAS Studio (the same programming environment as SAS University Edition) was used to write and test all the code shown in this book. The functions and call routines demonstrated are from Base SAS, SAS/STAT, SAS/GRAPH, and SAS/OR.

Example Code and Data

You can access the example code and data for this book from the author page at https://support.sas.com/authors. Look for the cover thumbnail of this book and select "Example Code and Data."

SAS University Edition

If you are using SAS University Edition to access data and run your programs, check the SAS University Edition page to ensure that the software contains the product or products that you need to run the code: www.sas.com/universityedition.

At the time of printing, everything in the book, with the exception of the code in chapter 7, can be run with SAS University Edition. The analysis performed in chapter 7 uses procedures that are available only through SAS/OR.

About the Author

Matthew Windham is a Principal Analytical Consultant in the SAS U.S. Government and Education practice, with a focus on Federal Law Enforcement and National Security programs. Before joining SAS, Matthew led teams providing mission-support across numerous federal agencies within the U.S. Departments of Defense, Treasury, and Homeland Security. Matthew is passionate about helping clients improve their daily operations through the application of mathematical and statistical modeling, data and text mining, and optimization. A longtime SAS user, Matthew enjoys leveraging the breadth of the SAS Platform to create innovative analytics solutions that have operational impact. Matthew is a Certified Analytics Professional, received his BS in Applied Mathematics from NC State University, and received his MS in Mathematics and Statistics from Georgetown University.

Learn more about this author by visiting his author page at https://support.sas.com/en/books/authors/matthew-windham.html. There you can download free book excerpts, access example code and data, read the latest reviews, get updates, and more.

We Want to Hear from You

SAS Press books are written *by* SAS Users *for* SAS Users. We welcome your participation in their development and your feedback on SAS Press books that you are using. Please visit sas.com/books to do the following:

- Sign up to review a book
- Recommend a topic
- Request information on how to become a SAS Press author
- Provide feedback on a book

Do you have questions about a SAS Press book that you are reading? Contact the author through saspress@sas.com or https://support.sas.com/author_feedback.

SAS has many resources to help you find answers and expand your knowledge. If you need additional help, see our list of resources: sas.com/books.

Chapter 1: Getting Started with Regular Expressions

1.1 Introduction

This chapter focuses entirely on developing your understanding of regular expressions (RegEx) before getting into the details of using them in SAS. We will begin actually implementing RegEx with SAS in Chapter 2. It is a natural inclination to jump right into the SAS code behind all of this. However, RegEx patterns are fundamental to making the SAS coding elements useful. Without my explaining RegEx first, I could discuss the forthcoming SAS functions and calls only at a very theoretical level, and that is the opposite of what I am trying to accomplish. Also, trying to learn too many different elements of any process at the same time can simply be overwhelming for you.

To facilitate the mission of this book—practical application—without overwhelming you with too much information at one time (new functions, calls, and expressions), I will present a short bit of test code to use with the RegEx examples throughout the chapter. I want to stress the point that obtaining a thorough understanding of RegEx syntax is critical for harnessing the full power of this incredible capability in SAS.

1.1.1 Defining Regular Expressions

Before going any further, we need to define *regular expressions*.

Taking the very formal definition might not provide the desired level of clarity:

Definition 1 (formal)
regular expressions: "Regular expressions consist of constants and operator symbols that denote sets of strings and operations over these sets, respectively."[1]

In the pursuit of clarity, we will operate with a slightly looser definition for regular expressions. Since practical application is our primary aim, it doesn't make sense to adhere to an overly esoteric definition. So, for our purposes we will use the following:

Definition 2 (informal, easier to understand)
regular expressions: character patterns used for automated searching and matching.

In SAS programming, regular expressions are seen as strings of letters and special characters that are recognized by certain built-in SAS functions for the purpose of searching and matching. Combined with other built-in SAS functions and procedures, you can realize tremendous capabilities, some of which we explore in the next section.

Note: SAS uses the same syntax for regular expressions as the Perl programming language.[2] Thus, throughout SAS documentation, you find regular expressions repeatedly referred to as "Perl regular expressions." In this book, I chose the conventions that the SAS documentation uses, unless the Perl conventions are the most common to programmers. To learn more about how SAS views Perl, see the SAS documentation online.[3] To learn more about Perl programming, see the Perl programming documentation.[4] In this book, however, I primarily dispense with the references to Perl, as they can be confusing.

1.1.2 Motivational Examples

The information in this book is very useful for a wide array of applications. However, that will not become obvious until after you read it. So, in order to visualize how you can use this information in your work, I present some realistic examples.

As you are probably familiar with, data is rarely provided to analysts in a form that is immediately useful. It is frequently necessary to clean, transform, and enhance source data before it can be used—especially

textual data. The following examples are devoid of the coding details that are discussed later in the book, but they do demonstrate these concepts at varying levels of sophistication. The primary goal here is to simply help you to see the utility for this information, and to begin thinking about ways to leverage it.

Extract, Transform, and Load (ETL)

ETL is a general set of processes for extracting data from its source, modifying it to fit your end needs, and loading it into a target location that enables you to best use it (e.g., database, data store, data warehouse). We're going to begin with a fairly basic example to get us started. Suppose we already have a SAS data set of customer addresses that contains some data quality issues. The method of recording the data is unknown to us, but visual inspection has revealed numerous occurrences of duplicative records, as in the table below. In this example, it is clearly the same individual with slightly different representations of the address and encoding for gender. But how do we fix such problems automatically for all of the records?

First Name	Last Name	DOB	Gender	Street	City	State	Zip
Robert	Smith	2/5/1967	M	123 Fourth Street	Fairfax,	VA	22030
Robert	Smith	2/5/1967	Male	123 Fourth St.	Fairfax	va	22030

Using regular expressions, we can algorithmically standardize abbreviations, remove punctuation, and do much more to ensure that each record is directly comparable. In this case, regular expressions enable us to perform more effective record keeping, which ultimately impacts downstream analysis and reporting.

We can easily leverage regular expressions to ensure that each record adheres to institutional standards. We can make each occurrence of Gender either "M/F" or "Male/Female," make every instance of the Street variable use "Street" or "St." in the address line, make each City variable include or exclude the comma, and abbreviate State as either all caps or all lowercase.

This example is quite simple, but it reveals the power of applying some basic data standardization techniques to data sets. By enforcing these standards across the entire data set, we are then able to properly identify duplicative references within the data set. In addition to making our analysis and reporting less error-prone, we can reduce data storage space and duplicative business activities associated with each record (for example, fewer customer catalogs will be mailed out, thus saving money!). For a detailed example involving ETL and how to solve this common problem of data standardization, see Section 2.4.1 in Chapter 2.

Data Manipulation

Suppose you have been given the task of creating a report on all Securities and Exchange Commission (SEC) administrative proceedings for the past ten years. However, the source data is just a bunch of .xml (XML) files, as shown in Figure 1.1. To the untrained eye, this looks like a lot of gibberish; to the trained eye, it looks like a lot of work.

Figure 1.1: Sample of 2009 SEC Administrative Proceedings XML File[5]

```
<?xml version="1.0" encoding="ISO-8859-1"?>
- <root>
  - <administrative_proceeding>
      <url>http://www.sec.gov/litigation/admin/2009/34-61262.pdf</url>
      <release_number>34-61262</release_number>
      <release_date>Dec. 30, 2009</release_date>
      <respondents>Stephen C. Gingrich</respondents>
    </administrative_proceeding>
  - <administrative_proceeding>
      <url>http://www.sec.gov/litigation/admin/2009/34-61256.pdf</url>
      <release_number>34-61256</release_number>
      <release_date>Dec. 30, 2009</release_date>
      <respondents>Gabelli Funds LLC</respondents>
    </administrative_proceeding>
  - <administrative_proceeding>
      <url>http://www.sec.gov/litigation/admin/2009/34-61255.pdf</url>
      <release_number>34-61255</release_number>
      <release_date>Dec. 30, 2009</release_date>
      <respondents>Gabelli Funds LLC</respondents>
    </administrative_proceeding>
```

However, with the proper use of regular expressions, creating this report becomes a fairly straightforward task. Regular expressions provide a method for us to algorithmically recognize patterns in the XML file, parse the data inside each tag, and generate a data set with the correct data columns. The resulting data set would contain a row for every record, structured similarly to this data set (for files with this transactional structure):

Example Data Set Structure

Release_Number	Release_Date	Respondents	URL
34-61262	Dec 30, 2009	Stephen C. Gingrich	http://www.sec.gov/litigation/admin/2009/34-61262.pdf
…	…	…	…

Note: Regular expressions cannot be used in isolation for this task due to the potential complexity of XML files. Sound logic and other Base SAS functions are required in order to process XML files in general. However, the point here is that regular expressions help us overcome some otherwise significant challenges to processing the data. If you are unfamiliar with XML or other tag-based languages (e.g., HTML), further reading on the topic is recommended. Though you don't need to know them at a deep level in order to process them effectively, it will save a lot of heartache to have an appreciation for how they are structured. I use some tag-based languages as part of the advanced examples in this book because they are so prevalent in practice.

Data Enrichment

Data enrichment is the process of using the data that we have to collect additional details or information from other sources about our subject matter, thus enriching the value of that data. In addition to parsing and structuring text, we can leverage the power of regular expressions in SAS to enrich data.

So, suppose we are going to do some economic impact analysis of the main SAS campus—located in Cary, NC—on the surrounding communities. In order to do this properly, we need to perform statistical analysis using geospatial information.

The address information is easily acquired from www.sas.com. However, it is useful, if not necessary, to include additional geo-location information such as latitude and longitude for effective analysis and reporting of geospatial statistics. The process of automating this is non-trivial, containing advanced programming steps that are beyond the scope of this book. However, it is important for you to understand that the techniques described in this book lead to just such sophisticated capabilities in the future. To make these techniques more tangible, we will walk through the steps and their results.

1. Start by extracting the address information embedded in Figure 1.2, just as in the data manipulation example, with regular expressions.

Figure 1.2: HTML Address Information

```
<p>World Headquarters<br>
SAS Institute Inc.<br>
100 SAS Campus Drive<br>
Cary, NC 27513-2414, USA<br>
Phone:919-677-8000<br>
Fax:919-677-4444<br>
</p>
```

Example Data Set Structure

Location	Address Line 1	Address Line 2	City	State	Zip	Phone	Fax
World Headquarters	SAS Institute Inc.	100 SAS Campus Drive	Cary	NC	27513-2414	919-677-8000	919-677-4444

2. Submit the address for geocoding via a web service like Google or Yahoo for free processing of the address into latitude and longitude. Type the following string into your browser to obtain the XML output, which is also sampled in Figure 1.3.
http://maps.googleapis.com/maps/api/geocode/xml?address=100+SAS+Campus+Drive,+Cary,+NC&sensor=false

Figure 1.3: XML Geocoding Results

```
- <geometry>
  - <location>
      <lat>35.8301733</lat>
      <lng>-78.7664916</lng>
    </location>
    <location_type>ROOFTOP</location_type>
  - <viewport>
    - <southwest>
        <lat>35.8288243</lat>
        <lng>-78.7678406</lng>
      </southwest>
    - <northeast>
        <lat>35.8315223</lat>
        <lng>-78.7651426</lng>
      </northeast>
    </viewport>
</geometry>
```

3. Use regular expressions to parse the returned XML files for the desired information—latitude and longitude in our case—and add them to the data set.

   Note: We are skipping some of the details as to how our particular set of latitude and longitude points are parsed. The tools needed to perform such work are covered later in the book. This example is provided here primarily to spark your imagination about what is possible with regular expressions.

Example Data Set Structure

Location	…	Latitude	Longitude
World Headquarters	…	35.8301733	-78.7664916

4. Verify your results by performing a reverse lookup of the latitude/longitude pair that we parsed out of the results file using https://maps.google.com/. As you can see in Figure 1.4, the expected result was achieved (SAS Campus Main Entrance in Cary, NC).

Figure 1.4: SAS Campus Using Google Maps

Now that we have an enriched data set that includes latitude and longitude, we can take the next steps for out the economic impact analysis.

Hopefully, the preceding examples have proven motivating, and you are now ready to discover the power of regular expressions with SAS. And remember, the last example was quite advanced—some sophisticated SAS programming capabilities were needed to achieve the result end-to-end. However, the majority of the work leveraged regular expressions.

1.1.3 RegEx Essentials

RegEx consist of letters, numbers, metacharacters, and special characters, which form patterns. In order for SAS to properly interpret these patterns, all RegEx values must be encapsulated by delimiter pairs—forward slash, /, is used throughout the text. (Refer to the test code in the next section). They act as the container for our patterns. So, all RegEx patterns that we create will look something like this: /pattern/.

For example, suppose we want to match the string of characters "Street" in an address. The pattern would look like /Street/. But we are clearly interested in doing more with RegEx than just searching for strings. So, the remainder of this chapter explores the various RegEx elements that we can insert into / / to develop rich capabilities.

Metacharacter

Before going any farther, some upcoming terminology should be clarified. *Metacharacter* is a term used quite frequently in this book, so it is important that it is clear what it actually means. A metacharacter is a character or set of characters used by a programming language like SAS for something other than its literal meaning. For example, \s represents a whitespace character in RegEx patterns, rather than just being a \ and the letter "s" that is collocated in the text. We begin our discussion of specific metacharacters in Section 1.3.

All nonliteral RegEx elements are some kind of metacharacter. It is good to keep this distinction clear, as I also make references to *character* when I want to discuss the actual string values or the results of metacharacter use.

Special Character

A *special character* is one of a limited set of ASCII characters that affects the structure and behavior of RegEx patterns. For example, opening and closing parentheses, (and), are used to create logical groups of characters or metacharacters in RegEx patterns. These are discussed thoroughly in Section 1.2.

RegEx Pattern Processing

At this juncture, it is also important to clarify how RegEx are processed by SAS. SAS reads each pattern from left to right in sequential *chunks*, matching each element (character or metacharacter) of the pattern in succession. If we want to match the string "hello", SAS searches until the first match of the letter "h" is found. Then, SAS determines whether the letter "e" immediately follows, and so on, until the entire string is found. Below is some pseudo code for this process, for which the logic is true even after we begin replacing characters with metacharacters (it would simply look more impressive).

Pseudo Code for Pattern Matching Process

```
START     IF POS = "h" THEN POS+1 NEXT ELSE POS+1 GOTO START
IF POS = "e" THEN POS+1 NEXT ELSE POS+1 GOTO START
   IF POS = "l" THEN POS+1 NEXT ELSE POS+1 GOTO START
   IF POS = "l" THEN POS+1 NEXT ELSE POS+1 GOTO START
   IF POS = "o" THEN MATCH=TRUE GOTO END ELSE POS+1 GOTO START
END
```

In this pseudo code, we see the START tag is our initiation of the algorithm, and the END tag denotes the termination of the algorithm. Meanwhile, the NEXT tag tells us when to skip to the next line of pseudo code, and the GOTO tag tells us to jump to a specified line in the pseudo code. The POS tag denotes the character position. We also have the usual IF, THEN, and ELSE logical tags in the code.

Again, this example demonstrates the search for "hello" in some text source. The algorithm initiates by testing whether the first character position is an "h". If it is not true, then the algorithm increments the character position by one—and tests for "h" again. If the first position is an "h", the character position is incremented, and the code tests for the letter "e". This continues until the word "hello" is found.

1.1.4 RegEx Test Code

The following code snippet enables you to quickly test new RegEx concepts as we go through the chapter. As you learn new RegEx metacharacters, options, and so on, you can edit this code in an effort to test the functionality. Also, more interesting data can be introduced by editing the `datalines` portion of the code. However, because we haven't yet discussed the details of how the pieces work, I discourage making edits outside the marked places in the code in order to avoid unforeseen errors arising at run time.

To keep things simple, we are using the DATALINES statement to define our data source and print the source string and the matched portion to the log. This should make it easier to follow what each new metacharacter is doing as we go through the text. Notice that everything is contained in a single DATA step, which does not generate a resulting data set (we are using _NULL_). The first line of our code is an IF statement that tests for the first record of our data set. The RegEx pattern is created only if we have encountered the first record in the data set, and is retained using the RETAIN statement. Afterward, the pattern reference identifier is reused by our code due to the RETAIN statement. Next, we pull in the data lines using the INPUT statement that assumes 50-character strings. Don't worry about the details of the CALL routine on the next line for now. We start writing SAS code in Chapter 2.

Essentially, the CALL routine inside the `RegEx Testing Framework` code shown below uses the RegEx pattern to find only the first matching occurrence of our pattern on each line of the `datalines` data. Finally, we use another IF statement to determine whether we found a pattern match. If we did, the code prints the results to the SAS log.

```
/*RegEx Testing Framework*/
data _NULL_;
if _N_=1 then
do;
    retain pattern_ID;
    pattern="/METACHARACTERS AND CHARACTERS GO HERE/"; /*<--Edit the pattern here.*/
    pattern_ID=prxparse(pattern);
end;
input some_data $50.;
call prxsubstr(pattern_ID, some_data, position, length);
if position ^= 0 then
   do;
      match=substr(some_data, position, length);
      put match:$QUOTE. "found in " some_data:$QUOTE.;
   end;
datalines;
Smith, BOB A.
ROBERT Allen Smith
Smithe, Cindy
103 Pennsylvania Ave. NW, Washington, DC 20216
508 First St. NW, Washington, DC 20001
650 1st St NE, Washington, DC 20002
3000 K Street NW, Washington, DC 20007
1560 Wilson Blvd, Arlington, VA 22209
1-800-123-4567
1(800) 789-1234
```

```
;
run;
```

Note: I have provided a jumble of data in the `datalines` portion of the code above. However, feel free to edit the data lines to thoroughly test each metacharacter as we go through this chapter.

Output 1.1 shows an example of the SAS log output provided by the previous code. For this example, I used merely the character string /Street/ for the pattern in order to create the output.

Output 1.1: Example Output Where pattern=/Street/

```
"Street" found in "3000 K Street NW, Washington, DC 20007"
```

The remaining information in this chapter provides a solid foundation for building robust, complex patterns in the future. Each element discussed is an independently useful building block for sophisticated text manipulation and analysis capabilities. Once we begin to combine these basic elements, we will create some very powerful analytic tools.

1.2 Special Characters

In addition to / (the forward slash), the characters ( ) | and \ (the backslash) are special and are thus treated differently than the RegEx metacharacters to be discussed later. Since some of these special characters are so fundamental to the structure of the RegEx pattern construction, we need to briefly discuss them first.

( )
: The two parentheses create logical groups of pattern characters and metacharacters—the same way they work in SAS code for logic operations. It is important to create logical groupings in order to construct more sophisticated patterns. Nesting the parentheses is also possible.

|
: The vertical bar represents a logical OR (much like in SAS). Again, the proper use of this element creates more sophisticated patterns. We will explore some interesting ways to use this character, starting with the example in Table 1.1. It is important to remember that the first item in an OR condition always matches before moving to the next condition.

\
: The backslash is a tricky one as it has a couple of uses. It is used as an integral component of many other metacharacters (examples abound in Section 1.3). Think about it as an initiator that tells SAS, "Hey, this is a metacharacter, not just some letter." But that's not all it does. Since the special characters defined above also appear in text that we might want to process, the backslash also acts as a blocker that tells SAS, "Hey, treat this special character as just a regular character." By using \, we can create patterns that include parentheses, vertical bars, backslashes, forward slashes, and more—we simply add a \ in front of each occurrence of all the special characters that we want to treat as characters. For example, if we want our pattern to include open and closed parentheses respectively, the pattern would contain \(\).

Since you haven't learned any RegEx metacharacters yet, let's revisit strings using some of these new concepts. Notice that we can already start to match useful patterns with the characters and special characters.

**Table 1.1: Examples Using (), |, and **

Usage	Matches
/(C|c)at/	"Cat" "cat"
/cat|mouse/	"cat" "mouse"
/((S|s)treet)|((R|r)oad)/	"Street" "street" "Road" "road"
/\(This\)|\(That\) /	"(This)" or "(That)"

Note: In Perl parlance, \ is known as an *escape character*. To avoid any unnecessary confusion, we will dispense with this lingo and just refer to it as the backslash. However, be prepared to see that term used quite a bit in the Perl literature and on community websites.

Now, there are some additional special characters that also need the backslash in front of them in order to be matched as normal characters. They are: { } [ ] ^ $. * + and ?. All these characters are reserved and are thus treated differently, because they each have a special purpose and meaning in the world of RegEx. Since each one is defined and discussed at length in Sections 1.4 and 1.5, we will not discuss them further here. For now, just remember that they can't be used as part of pattern strings without the backslash immediately preceding them. Table 1.2 shows a few examples of how to use them as normal characters.

Table 1.2: Examples Using { } [ ] ^ $. * +

Usage	Matches
/\$1\.00 \+ \$0\.50 = \$1\.50/	"$1.00 + $0.50 = $1.50"
/2*3 = 6/	"2*3 = 6"
/\[2\]\^2/	"[2]^2"
/\{1,2,3,4,5\}/	"{1,2,3,4,5}"

Note: Notice that = and , match as characters (i.e., without a backslash) because they are not considered special characters.

1.3 Basic Metacharacters

As you write RegEx patterns in the future, you will find yourself using most of the metacharacters discussed in this section frequently because they are fundamental elements of RegEx pattern creation. Now, we can already build some useful patterns with the information discussed in Section 1.1. However, the metacharacters in this section create the greatest return on time investment due to how flexible and powerful they can make RegEx patterns.

Notice as we go through the examples how we can obtain some unexpected results. It is important to be very strategic when using some of these RegEx metacharacters as you don't always know what to expect in the text that you are processing. Even when you know the source quite well, there are inevitably errors or unknown changes that can wreck a poorly designed pattern. So, like any good analyst, you need to be thinking a few steps ahead in order to maintain robust RegEx code.

Note: Unlike SAS, all RegEx metacharacters are case sensitive, as you will see shortly. If a letter is defined here as lowercase or uppercase, then it MUST be used that way. Otherwise, your programs will do something

very different from what you expect. In other words, even though you can be lazy with capitalization when writing SAS code (e.g., DATA vs. data), the same is not true here.

1.3.1 Wildcard

The wildcard metacharacter, which is a period (.), matches any single character value, except for a newline character (\n). The ability to match virtually any single character will prove useful when you are searching for the superset of associated character strings. You might also want to use it when you have no idea what values might be in a particular character position. Table 1.3 provides examples.

Table 1.3: Examples Using .

Usage	Matches
/R.n/	"Ran" "Run" "R+n" "R n" "R(n" "Ron" …
/.un/	"Fun" "fun" "Run" "run" "bun" "(un" "-un" …
/Street./	"Street." "Street," "Streets" "Street+" "Street_"…

Note: The period matches anything except the newline character (\n)—including itself. This can be helpful, but must be used wisely. Also note, only \n matches the newline character.

1.3.2 Word

The metacharacter \w matches any word character value, which includes alphanumeric and underscore (_) values. It matches any single letter (regardless of case), number, or underscore for a single character position. But do not be fooled by the underscore inclusion; \w does NOT match hyphens, dashes, spaces, or punctuation marks. Table 1.4 provides examples.

Table 1.4: Examples Using \w

Usage	Matches
/R\wn/	"Ran" "Run" "Ron" …
/\wun/	"Fun" "fun" "Run" "run" "Bun" "bun" "_un" …
/Street\w/	"Streets" "Street_"

Note: The \w wildcard should not have any unintentional spaces before or after it. Such spaces result in the pattern trying to match those additional spaces in addition to the \w. (This goes for any RegEx metacharacter.)

1.3.3 Non-word

The metacharacter \W matches a non-word character value (i.e., everything that \w doesn't include, except for the ever-elusive \n). The \W metacharacter is valuable when you are unsure what is in a character cell but you know that you don't want a word character (i.e., alphanumeric and _). Table 1.5 provides examples.

Table 1.5: Examples Using \W

Usage	Matches
/Washington\W/	“Washington.” “Washington,” “Washington;”…
/D\WC\W/	“D.C.” “D,C.” “D C.” “D C “ …
/Street\W/	“Street.” “Street,” “Street+” …

Note: You will continue to see lowercase and uppercase versions of these RegEx characters acting as near opposites, with some exceptions. It might not be overly clever, but does help simplify matters.

1.3.4 Tab

The metacharacter \t matches only the tab character in a string. Unlike the RegEx characters to follow, this metacharacter matches only the tab whitespace character. This is especially useful when the tab holds some special significance, such as when you are processing tab-delimited text files. Table 1.6 provides examples.

Table 1.6: Examples Using \t

Usage	Matches
/SAS\t/	“SAS ”
/SAS\tInstitute\tInc/	“SAS Institute Inc”
/Street\t/	“Street ”

Note: This metacharacter does not have an opposite (i.e., \T does not exist).

1.3.5 Whitespace

The metacharacter \s matches on a single whitespace character, which includes the space, tab, newline, carriage return, and form feed characters. You must include this when you are matching on anything in text that is separated by white space, and you are unsure of which will occur. Table 1.7 provides examples.

Table 1.7: Examples Using \s

Usage	Matches
/SAS\s/	“SAS ” “SAS ”
/SAS\sInstitute\sInc/	“SAS Institute Inc” “SAS Institute Inc”
/Street\s/	“Street ” “Street “

Note: This form of the \s metacharacter matches only one whitespace character. We review how to find multiple matches in Section 1.5.2 because that is frequently needed when you are matching text.

1.3.6 Non-whitespace

The metacharacter \S matches on a single non-whitespace character—the exact opposite of \s. This metacharacter is often used to account for unexpected dashes, apostrophes, commas, and so on, that might otherwise prevent a match. Table 1.8 provides examples.

Table 1.8: Examples Using \S

Usage	Matches
/Leonato\Ss/	"Leonato's" "Leonatoas" "Leonato_s" …
/Washington\S/	"Washingtons" "Washington." "Washington," …
/Street\S/	"Street." "Street," "Streets" "Street+" "Street_"…

1.3.7 Digit

The metacharacter \d matches on a numerical digit character (i.e., 0–9). This RegEx metacharacter is probably the most straightforward one as it has a very narrow focus. Just remember that a single occurrence of \d is for only one character position in any text. In order to capture larger numbers (i.e., anything greater than 9), you have to build patterns with multiple occurrences of \d. Table 1.9 provides examples, but we discuss more sophisticated methods for accomplishing this later in the chapter. (See "Repetition Modifiers" in Section 1.5.2.)

Table 1.9: Examples Using \d

Usage	Matches
/\dst/	"1st" "9st" "4st" …
/10\d/	"101" "102" "103" …
/1-800-\d\d\d-\d\d\d\d/	"1-800-123-4567" "1-800-789-3456" …

Note: Just remember that even though your pattern might be correct, the data is not necessarily correct (4st and 9st don't make sense!).

1.3.8 Non-digit

The metacharacter \D matches on any single non-digit character. Again, this is the opposite of the lowercase metacharacter \d. This metacharacter matches on every value that is not a number. Table 1.10 provides examples.

Table 1.10: Examples Using \D

Usage	Matches
/1\D800\D123\D4567/	"1-800-123-4567" "1.800.123.4567" …
/1560\DWilson\DBlvd/	"1560 Wilson Blvd" "1560_Wilson_Blvd" …
/19\D\D\DStreet/	"19th Street" "19th.Street" "19…Street" …

1.3.9 Newline

The metacharacter \n matches a newline character. It is quite useful for some patterns to know that you have encountered a new line. For instance, you might be processing addresses in a text file, which often contain different pieces of information on different lines. Table 1.11 provides examples.

Table 1.11: Examples Using \n

Usage	Matches
/103 Pennsylvania Ave\. NW,\nWashington, DC 20216/	"103 Pennsylvania Ave. NW, Washington, DC 20216"
/<html tag>\n/	"<html tag> " …
/v\ne\nr\nt\ni\nc\na\nl\nt\ne\nx\nt/	"v e r t i c a l t e x t" …

Note: The test code does not enable us to actually try this metacharacter because it uses data lines, which is a feature of SAS that intentionally ignores newline characters when typed (i.e., pressing the Enter key just creates the start of a new data line in the SAS code window). For this reason, newline characters are not present in data lines for you to read and match on. But have faith, for now, that this one works as advertised. You will discover ways to process different text sources in the next chapter, enabling you to process newline characters.

1.3.10 Bell

The metacharacter \a matches an alarm "bell" character. The alarm character falls into a class of non-printing or invisible characters that are part of the ASCII character set. ASCII was developed long ago when operating systems used non-printing characters fairly extensively. Today, however, these characters are relatively uncommon, and most often occur only in files meant for computers to read rather than humans—since they are not displayed. When encountered, these characters generate an alarm tone, or "bell," on a computer's internal speaker. While they are often associated with errors, they can also be used to alert users that the end of a file or process has been achieved (e.g., in a system log file). You can use this metacharacter when you know to expect such a character in a source file. Table 1.12 provides examples.

Table 1.12: Examples Using \a

Usage	Matches
/\a END OF FILE/	"**BEL** END OF FILE"
/PROCESS COMPLETED SUCCESSFULLY\a/	"PROCESS COMPLETED SUCCESSFULLY **BEL**" …
/\aERROR/	"**BEL**ERROR" …

Note: Since the alarm character is a non-printing ASCII character, I am representing its location in the matching text with the BEL ASCII character. However, remember that such a code does not appear in our text.

1.3.11 Control Character

The metacharacter \cA-\cZ matches a control character for the letter that follows the \c. For example, \cF matches control-F in the source. This is one of several examples where you might be processing less-often-used file types (i.e., not a file meant for humans to read). Control characters, or non-printing characters, were once used extensively by transactional computing and telecommunications systems. These control characters, while not visible in most text editors, are still part of the ASCII character set, and can still be used by older systems in these regimes. For our examples in Table 1.13, we stick with the convention that is used for the alarm metacharacter above—the standard ASCII abbreviation is used despite the fact that they are never actually seen in text.

Table 1.13: Examples Using \cA-\cZ

Usage	Matches
/\cP/	**DEL** the non-printing **Data Link Escape** ASCII control character ^P
/\cB/	**STX** the non-printing **Start of Text** ASCII control character ^B
/\cBhello\cC/	**STX**hello**ETX** the non-printing **Start of Text** ASCII control character ^B followed by the character string "hello" and completed with the non-printing **End of Text** ASCII control character ^C

1.3.12 Octal

The metacharacter \ddd matches an octal character of the form *ddd*.[6] It is used to match on the octal code for an ASCII character for which you are searching. It can be especially useful when you need to find specific non-printing ASCII characters in a file. The default behavior by SAS is to return the ASCII character associated with this octal code in the results. Table 1.14 provides examples.

Table 1.14: Examples Using \ddd

Usage	Matches	Notes
/\s\041\s/	" ! "	This octal code translates to the ! ASCII character.
/\110\105\114\114\117/	"HELLO"	This series of octal codes translate to the "HELLO" string of ASCII characters.

Usage	Matches	Notes
/\s\007\011\s/	" **BELTAB** "	These octal codes translate to the two non-printing ASCII characters **BEL** and **TAB**. Refer to our discussion of the alarm metacharacter in Section 1.3.10 regarding characters that are not displayed.

Note: You will discover how to search for ranges of these values in the next section (Section 1.4). Also note that the largest ASCII value is decimal 127, octal 177, and hexadecimal 7F.

1.3.13 Hexadecimal

The metacharacter \xdd matches a hexadecimal character of the form *dd*.[7] The purpose of our implementation here is again not about searching through raw hexadecimal files, etc. We are using this to search for the hexadecimal code associated with the ASCII characters that we want in a source (manipulation of raw hex data sources is a different book). Table 1.15 provides examples.

Table 1.15: Examples Using \xdd

Usage	Matches	Notes
/\x2B/	"+"	This hexadecimal code translates to the + ASCII character.
/\x31\x2B\x31\x3D\x32/	"1+1=2"	These hexadecimal codes translate to the 1+1=2 ASCII characters.
/\x30\x30\x20\x46\x46/	"00 FF"	This is a reminder that we can match hexadecimal numbers stored in ASCII, and that they are not the same.

1.4 Character Classes

In addition to using the built-in RegEx characters to match patterns, users have the ability to create custom character matching. This capability is derived via different uses of [and] (square braces). The square braces essentially create a custom metacharacter, where the items contained between the opening brace and closing brace are possible match values for a single character cell. In addition to putting a list characters inside the braces, you can also include metacharacters. Each metacharacter discussed below includes an example, which includes the use of a metacharacter, and they all have the same match results. Just for fun, they are all identifying a hexadecimal number range present in the ASCII source file (stored as ASCII characters in the source file, but representing the range of possible hexadecimal values).

Note: Remember that some of the components discussed in this section are special characters that must be escaped with \ in order to be matched in isolation. Specifically, these characters are: ^, [, and].

1.4.1 List

The metacharacter […] matches any one of the specific characters or metacharacters listed within the braces. Being able to define an unordered list of things that you want to appear in a space is very

convenient, and can sometimes be more convenient than the metacharacters that identify broad classes of character types. Table 1.16 provides examples.

Table 1.16: Examples Using […]

Usage	Matches
/[abcABC]/	"a" "b" "c" "A" "B" "C"
/[0173]/	"0" "1" "3" "7"
/[CcBbRr]at/	"cat" "Cat" "bat" "Bat" "rat" "Rat"
/[\dABCDEF]/	"0" "1" "2" "3" "4" "5" "6" "7" "8" "9" "A" "B" "C" "D" "E" "F"

1.4.2 Not List

The metacharacter [^…] matches one of anything not listed within the braces, except for the newline character. Sometimes it is easier to write down what we don't want rather than what we do. And for that reason, we might want to use this metacharacter. We can quickly identify the unwanted items and define them here. Table 1.17 provides examples.

Table 1.17: Examples Using [^…]

Usage	Matches
/[^abcABC]/	"d" "e" "f" …
/[^0173]/	"2" "4" "5" "6" "8" "9"
/[^Cc]at/	"fat" "Fat" "hat" "rat" "mat" "Hat" …
/[^\WGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz_]/	"0" "1" "2" "3" "4" "5" "6" "7" "8" "9" "A" "B" "C" "D" "E" "F"

1.4.3 Range

The metacharacter […-…] matches anything that falls into a range of character values. In other words, case matters for letters listed in the braces. RegEx, and by extension SAS, understands the inherent order of letters and numbers. Therefore, we can define any range of numbers or letters to be matched by this metacharacter. Table 1.18 provides examples.

Table 1.18: Examples Using […-…]

Usage	Matches
/[f-m]/	"f" "g" "h" "i" "j" "k" "l" "m"
/[1-9]/	"1" "2" "3" "4" "5" "6" "7" "8" "9"
/[a-cA-C]/	"a" "b" "c" "A" "B" "C"
/[\dA-F]/	"0" "1" "2" "3" "4" "5" "6" "7" "8" "9" "A" "B" "C" "D" "E" "F"

1.5 Modifiers

There are two significant things that you probably noticed as missing from the previous sections, which are worth further discussion here. First, all of the applicable metacharacters thus far have ignored letter case. In other words, \w, \S, \D, and . all match on a letter regardless of whether it is lowercase or uppercase. However, there are situations in which the case of a letter becomes important, but the letter itself is not known in advance.

Second, we can use a single match character as many times as we like, which creates additional fuzziness for our matches. However, there is a downside to just typing them out: *each occurrence must exist in order to match the pattern*. For instance, if the source text for the \D examples above contained "19thStreet" with no spaces, we'd never find it by using \D three times. And since the primary goal of the RegEx capability is to have automated text processing, we need a robust way to make this kind of matching more flexible.

Over the next two subsections (1.5.1 and 1.5.2), we will work through ways to overcome these limitations by using modifiers. There are two types of modifiers, case modifiers and repetition modifiers. Combining them gives us significant robustness and flexibility in real-world RegEx implementations, and should be considered as fundamental to real-world implementations as the metacharacters that we have discussed thus far.

1.5.1 Case Modifiers

When performing matches on text, there is the obvious consideration of letter case (upper vs. lower). Although I have already introduced a rudimentary way to handle this in situations where the letter is known, there still must be a methodology for accounting for letter case when it is unknown. This section discusses a variety of approaches to dealing with case matching. Depending on the situation, some approaches are more convenient than others, while not necessarily being right or wrong.

Lowercase

The metacharacter \l matches when the next character in a pattern is lowercase. This metacharacter applies only to characters (metacharacters, groups, and so on, don't work). In practice, it is more practical to simply type the lowercase version of the desired character value, or provide a list of lowercase letters to match. Table 1.19 provides examples.

Table 1.19: Examples Using \l

Usage	Matches
/\lStreet/	"street" …
/\s\lS\lA\lS\sInstitute/	" sas Institute" …
/(\lS|\lF)leet/	"sleet" "fleet" …

Uppercase

The metacharacter \u matches when the next letter in a pattern is uppercase. It functions exactly as the lowercase version introduced above (\l), but also applies to uppercase. Table 1.20 provides examples.

Table 1.20: Examples Using \u

Usage	Matches
/\uinc./	"Inc." …
/\ustreet|\ust\./	"Street" "St." …
/\uave\.|\uavenue,/	"Ave." "Avenue,"

Lowercase Range

The metacharacter \L…\E matches when all the characters between the \L and \E are lowercase. Strings typed between \L and \E are forced to match on lowercase only, even when they are typed in as capital letters. However, unlike the \l metacharacter, \L…\E can also contain character classes and repetition modifiers. Table 1.21 provides examples.

Table 1.21: Examples Using \L…\E

Usage	Matches
/\L[a-z0-9][a-z0-9][a-z0-9]\E/	"sas" "abc" "123" …
/\LTHESE ARE LOWERCASE\E/	"these are lowercase"
/\sR\L[a-z][a-z][a-z]\E\s/	" Read " " Road " " Rode " " Ride " " Real " …

Note: When applying case modifiers to non-alphabet characters, the modifier is ignored. It doesn't apply to those characters, so it doesn't affect the match.

Uppercase Range

The metacharacter \U…\E creates a match when all the characters between the \U and \E are uppercase. Again, this metacharacter functions the same way as the lowercase version discussed above, but applies to uppercase. This metacharacter can be useful for identifying acronyms or other text where capital letters are important. Table 1.22 provides examples.

Table 1.22: Examples Using \U…\E

Usage	Matches
/\U[a-z][a-z][a-z]\E/	"SAS" "CIA" …
/\U[a-z][a-z][a-z]\E\sInstitute\sInc\W/	"SAS Institute Inc." …
/\s\Uallcaps\E\s/	" ALLCAPS "

Note: Notice that other metacharacters are not allowed inside \L…\E or \U…\E metacharacters. In other words, \w can't be used to replace the character classes above.

Quote Range

The metacharacter \Q…\E matches all content inside the \Q and \E as character strings, disabling everything including the backslash character. Metacharacters cannot be used inside \Q…\E. The functionality provided by this metacharacter is great for searching within strings that contain a significant number of reserved characters, such as XML, webserver logs, or HTML. Table 1.23 provides examples.

Table 1.23: Examples Using \Q…\E

Usage	Matches
/\Q<html tag name>\E/	"<html tag name>"
/\Qf(x) + f(y) = z\E/	"f(x) + f(y) = z"
/\Q<!DOCTYPE HTML> <html lang="en-US">\E/	"<!DOCTYPE HTML> <html lang="en-US">"

1.5.2 Repetition Modifiers

Repetition modifiers change the matching repetition behavior of the metacharacters and characters immediately preceding them in a pattern. They can also modify the matching repetition of an entire group—defined using () to surround the group of metacharacters and characters before the modifier. Just keep in mind that repetition of the entire group means that it repeats back-to-back (e.g., "haha"), unless we also modify the individual metacharacters.

Now, there are two types of repetition modifiers, *greedy* and *lazy*. Greedy repetition modifiers try to match as many times as possible within the confines of their definition. Lazy modifiers attempt to find a match as few times as possible. They have similar uses, which can make the difference between their results subtle.

Introduction to Greedy Repetition Modifiers

Let's start by discussing greedy modifiers because they are a little more intuitive to use. As we go through the examples, it is important to keep in mind that greedy modifiers match as many times as possible—constantly searching for the last possible time the match is still true. It is therefore easy to create patterns that match differently from what you might expect.

There is a concept in RegEx known as *backtracking*, which is the root cause for potential issues with greedy modifiers (hint: backtracking results in the need for lazy modifiers). As we discuss further when we examine lazy repetition modifiers, a greedy modifier actually tries to maximize the matches of a modified pattern chunk by searching until the match fails. Upon that failure, the system then *backtracks* to the position where the modified chunk last matched. The processing time wasted with backtracking for a single match is insignificant. However, as soon as we introduce a few additional factors, this problem can waste tremendous computing cycles—multiple modified pattern chunks, numerous match iterations (think loops), and large data sources. It is important to be mindful of these factors when designing patterns as they can have unintended consequences.

Greedy 0 or More

The modifier * requires the immediately preceding character or metacharacter to match 0 or more times. It enables us to generate unlimited optional matches within text. For example, we might want to match every occurrence of a word root, along with all of its prefixes and suffixes. By allowing the prefixes and suffixes to be optional, we are able to achieve this goal. Table 1.24 provides examples.

Table 1.24: Examples Using *

Usage	Matches
/Sing\w*/	"Sing" "Sings" "Singing" "Singer" "Singers" …
/D\W*C\W*/	"DC" "D.C." "D C " "D….-!$%^ C.-)*&^%"…
/19\D*Street/	"19th Street" "19thStreet" "19Street" …
/Hello*/	"Hell" "Hello" "Hellooooooooooooo" …

Greedy 1 or More

The modifier + requires the immediately preceding character or metacharacter to match 1 or more times. The plus sign modifier works similarly to the asterisk modifier, with the exception that it enforces a match of the metacharacter or character at least 1 time. Table 1.25 provides examples.

Table 1.25: Examples Using +

Usage	Matches
/Ru\w+/	"Run" "Ruin" "Runt" "Runners" …
/\s\U[a-z]+\E\s/	Words with all letters capitalized, and surrounded by spaces.
/19\D+Street/	"19th Street" "19th.Street" "19…Street" …
/(ha)+/	"ha" "hahahahahahaha" …

Note: Pay special attention to the addition of the \s metacharacter in the second example in Table 1.25. If it were not present, the pattern would also match only single capital letters at the beginning of words. By adding \s, the pattern requires a whitespace character to immediately follow the one or more capital letters, thus eliminating matches on single letters at the beginning of words.

Greedy 0 or 1 Time

The modifier ? creates a match of only 0 or 1 time. The question mark provides us the ability to make the occurrence of a metacharacter optional without allowing it to match multiple times. This can be effective for matching word pairs that have inconsistent use of dashes or spaces (e.g., short-term vs. short term). Table 1.26 provides examples.

Table 1.26: Examples Using ?

Usage	Matches
/1\D?800\D?123\D?4567/	"1-800-123-4567" "18001234567" …
/1560\sWilson\sBlvd\W?/	"1560 Wilson Blvd." "1560 Wilson Blvd" …
/19th\s?Street/	"19th Street" "19thStreet" …

Greedy n Times

The modifier {n} creates a match of exactly *n* times. Being able to match on a metacharacter exactly *n* number of times is the same as typing that metacharacter out that many times. However, from the perspective of coding and maintaining the RegEx patterns, using the modifier is a much better approach. It limits the opportunity for us to make typographical errors when initially creating the RegEx pattern, and it improves readability when later editing and sharing the patterns. Table 1.27 provides examples.

Table 1.27: Examples Using {n}

Usage	Matches
/1-800-\d{3}-\d{4}/	"1-800-123-4567" "1.800.123.4567" …
/R\w{4}/	"Round" "Runts" "Ruins" …
/19\D{3}Street/	"19th Street" "19th.Street" "19…Street" …
/(\d{5}-\d{4})|(\d{5})/	"12345-6789" "12345" …

Greedy n or More

The modifier {n,} creates a match at least *n* times. By ensuring that we can match something at least *n* times, we are able to create functionality very similar to the plus modifier. However, we are raising the minimum number of times that the metacharacter must match. This is quite useful for certain applications, but must be handled with caution. Also, like the + modifier, we can easily get very long strings of unanticipated matches due to a single logical error in pattern construction. Table 1.28 provides examples.

Table 1.28: Examples Using {n,}

Usage	Matches
/1-800-\d{1,}-\d{2,}/	"1-800-123-4567" "1-800-789-12" …
/\d{3,}-\d{2,}-\d{4,}/	"143-25-7689" "12345689-546545654-9820"…
/19\D{3,}Street/	"19th Street" "19th, Not My Street" …

Note: Be mindful not to type a space after the comma inside the curly braces. It is easy to do out of habit, but it will wreck our pattern!

Greedy n to m Times

The modifier {n,m} creates a match at least *n*, but not more than *m* times. Creating a match with a specified range is quite useful for ensuring that data quality standards are being maintained. When extracting semi-structured data elements such as ZIP codes, birthdates, and phone numbers, it is important to maintain a certain level of flexibility while also ensuring that the source is within expected tolerances. For instance, a two-digit year might be accepted in lieu of a four-digit year, but a four-digit zip would be unacceptable. Table 1.29 provides examples.

Table 1.29: Examples Using {n,m}

Usage	Matches
/(1-)?8\d\d-\d{3,3}-\d{4,4}/	"1-800-123-4567" …
/\d{1,2}-\d{1,2}-\d{2,4}/	"10-20-1950" "8-30-52" "4-3-1979"…
/Was{1,7}/	"Washington" "Wash" "Waste" "Washing" …

Note: As you can see in the examples above, the {n,m} might not always be the best choice of modifier, but these examples are meant to demonstrate the flexibility of implementation. For instance, the year in the second example is allowed to be three digits with this usage. Using an OR clause with the {n} modifier is a simple fix.

Introduction to Lazy Repetition Modifiers

Now that you are familiar with greedy modifiers, let's begin examining the lazy ones. In terms of syntax, they differ from the greedy modifiers only by the addition of a question mark (?). By adding the question mark immediately after each of the greedy modifiers, we are able to subtly change their behavior—sometimes in unexpected ways.

In general, lazy modifiers are used to both avoid overmatching and improve performance when compared to the greedy modifiers. There are situations when matching with greedy modifiers would lead to either grabbing too much information, or simply slowing down system performance. For instance, processing semi-structured text files such as HTML or XML is a great example of when lazy modifiers would come in handy.

Lazy 0 or More

The modifier *? creates a match 0 or more times, but as few times as necessary to create the match. In some situations, it creates the same matches as does the greedy version. However, in other cases, the results are very different. To make it clearer, Table 1.30 describes the details of a few examples.

Table 1.30: Examples Using *?

Usage	Matches	Notes
/Sing\w*?/	"Sing"	This matches only the word "Sing" because the modifier is given the option to match nothing. And since it is *lazy*, it will take that option every time, regardless of whether a word character immediately follows the "g" in "Sing".
/Sing\w*?\s/	"Singing " …	Comparing this to the example above, you see that appending the \s on the pattern creates additional matches. The \s forces the pattern to continue searching for a match that includes white space. This could be "Sing " or many other combinations (similar to the greedy outcomes).
/(ha)*?/	""	This example demonstrates why we need to be careful with lazy modifiers. Even when "ha" exists, it is ignored, again because the modifier has the option to do so. The greedy version of this would match as many times as the word "ha" occurred back-to-back, with a minimum of zero times.

Lazy 1 or More

The modifier +? creates a match 1 or more times, but as few times as necessary to create a match. Again, if it is possible, this matches only once. Table 1.31 provides examples.

Table 1.31: Examples Using +?

Usage	Matches	Notes
/Sing\w+?/	"Singi"	This matches only "Sing" plus exactly one word character following the "g". Again, by giving the lazy modifier an option to match the minimum, it will do so every time.
/Sing\w+?\s/	"Singing " …	Again, we see that appending the \s on the pattern creates additional matches. The \s forces the pattern to continue searching for a match that includes white space. This could be "Singi " or many other combinations (similar to the greedy outcomes).
/(ha)+?/	"ha"	This example is less of a cautionary tale than for *?. But it might still provide undesirable results. Even when "ha" exists numerous times back-to-back, it matches only the first time, unless an additional match element follows it. Again, this is because the modifier has the option to match only once. The greedy version of this would match as many times as the word "ha" occurred back-to-back, with a minimum of once.

Lazy 0 or 1 Times

The modifier ?? creates a match 0 or 1 times, but as few times as necessary to create a match. Unless forced, this modifier will match 0 times. Table 1.32 provides examples.

Table 1.32: Examples Using ??

Usage	Matches	Notes
/Sing\w??/	"Sing"	This matches only the word "Sing" because the modifier is given the option to match nothing. And since it is *lazy*, it will take that option every time, regardless of whether a word character immediately follows the "g" in "Sing". The reasoning is the same as with the *? modifier.
/Sing\w??\s/	"Sings " …	Again, just as with the *? modifier, we see that appending the \s on the pattern creates additional matches. The \s forces the pattern to continue searching for a match that includes white space. This could be "Sings " or a few other combinations (similar to the greedy outcomes).
/(ha)??/	""	This example demonstrates why we need to be careful with lazy modifiers. Even when "ha" exists, it is ignored, again because the modifier has the option to do so. The greedy version of this would match as many times as the word "ha" occurred back-to-back.

Lazy n Times

The modifier {n}? creates a match exactly *n* times. This modifier functions exactly as the greedy version, making the ? unnecessary. Using this modifier results in no performance enhancement or change in functionality, which makes it a completely unnecessary addition to the Perl language. It has been included here for the sake of completeness. Table 1.33 shows that the same examples reveal the same results.

Table 1.33: Examples Using "{n}?"

Usage	Matches
/1-800-\d{3}?-\d{4}?/	"1-800-123-4567" "1.800.123.4567" …
/R\w{4}?/	"Round" "Runts" "Ruins" …
/19\D{3}?Street/	"19th Street" "19th.Street" "19…Street" …
/(\d{5}?-\d{4}?)|(\d{5}?)/	"12345-6789" "12345" …

Lazy n or More

The modifier {n,}? creates a match, at least *n* times and as few times as necessary to create a match. This functions just like the *? or +? modifiers, except that the minimum number of matches is arbitrary. Again, we see similar behavior resulting from the laziness of the modifier. Table 1.34 provides examples.

Table 1.34: Examples Using {n,}?

Usage	Matches	Notes
/Sing\w{3,}?/	"Singing" …	This usage matches exactly *n*=3 times. Again, by giving the lazy modifier an option to match the minimum, it will do so every time.
/0{3,}?\s/	"0000 " …	Now that you have the hang of these modifiers, this example should be a little more interesting. Appending \s on the pattern still forces it to match each 0 until the white space is encountered. The pattern is "anchored" to the first occurrence of a 0, thus capturing more than the minimum.
/(ha){4,}?/	"hahahaha"	Without surrounding information in the pattern, this matches only the minimum number of times. By having nothing else to force additional matching, the lazy modifier just stops after the minimum of *n*=4.

Lazy n to m Times

The modifier {n,m}? creates a match at least *n* times, but no more than *m* times—as few times in that range as necessary to create the match. It functions like many of the other lazy modifiers discussed thus far, but it sets a cap on how many times it can match in addition to having an arbitrary minimum. Table 1.35 provides examples.

Table 1.35: Examples Using {n,m}?

Usage	Matches	Notes
/Read\w{1,3}?/	"Ready" …	This usage matches the word metacharacter only one time. Again, by giving the lazy modifier an option to match the minimum, it will do so every time.
/0{2,5}?\s/	"0000 " …	Again, the pattern is "anchored" to the first occurrence of a 0, thus capturing the minimum if it exists, up to the maximum.
/\sha(ha){0,6}?/	" ha"	By not having anything after the "anchor" point for the pattern to match on, there is nothing to force additional matching. The lazy modifier just stops after the minimum of *n*=0.

1.6 Options

Options affect the behavior of the entire RegEx pattern with which they are associated. These behavioral changes provide benefits ranging from making RegEx creation more convenient, to providing new or enhanced functionality.

Options occur *after* the closing slash character, but there is one item of significance that occurs *before* the first slash character that we will also discuss—it is not actually an option but this is best place to go over it. And we are not going to cover all of the options for the same reason we haven't covered absolutely all of the metacharacters thus far—this is an introductory text.

1.6.1 Ignore Case

The option //i ignores letter case for the entire pattern, even character strings. This is a great option to use when we know exactly what words we are searching for, but we don't want the letter case to be an issue. Table 1.36 provides examples.

Table 1.36: Examples Using //i

Usage	Matches
/1600 Pennsylvania Avenue/i	"1600 pennsylvania avenue" "1600 PENNSYLVANIA AVENUE" …
/STREET/i	"street" "Street" "STREET" …
/CAPS don't MaTtEr/i	"caps don't matter" "CaPs DoN't MATTER" …

1.6.2 Single Line

The option //s forces the dot character (.) to match everything, including the newline character, when it occurs in the pattern. This can be very helpful to ensure that we don't miss anything for a particular character position. Table 1.37 provides examples.

Table 1.37: Examples Using //s

Usage	Matches
/43rd and Times Square.New York, NY 10036/s	"43rd and Times Square New York, NY 10036" …
/Bob Smith.\d{3}-\d{3}-\d{4}/s	"Bob Smith 123-456-7891" …

1.6.3 Multiline

The option //m causes ^ and $ to match on more than just the string start and end respectively. Instead, they match on every newline encountered because the various lines of information are treated as one continuous line. This enhanced functionality really applies to two metacharacters that we haven't covered yet (we'll discuss them in Section 1.7), so if you need to, feel free to peek ahead and come back to this one. Table 1.38 provides examples.

Table 1.38: Examples Using //m

Usage	Matches
/^\w+/m	Words at the beginning of a string and words following a newline character.
/\w+?\s$/m	Words immediately before a space and the string end, and before a space and newline character.

1.6.4 Compile Once

The option //o is known as the *compile once* option. By having the "o" immediately following the closing slash, SAS knows to compile that RegEx only once. This option creates a very nice simplification to SAS code, which I demonstrate by showing updated test code below (see Section 1.1.4 for the original code). Notice how the IF block is removed, and only the two lines that do not include the RETAIN statement remain. These changes are possible due to the compilation happening the first time through the DATA step. Every subsequent loop through reuses the previously compiled expression, if it exists.

Updated Test Code

```
/*RegEx Testing Framework*/
data _NULL_;
*if _N_=1 then
*do;
*   retain pattern_ID;
*   pattern="/Run/"; /*<--Edit the pattern here.*/
*   pattern_ID=prxparse(pattern);
*end;
```

```
pattern="/Run/o"; /*<--Edit the pattern here.*/
pattern_ID=prxparse(pattern);
input some_data $50.;
call prxsubstr(pattern_ID, some_data, position, length);
if position ^= 0 then
   do;
      match=substr(some_data, position, length);
      put match:$QUOTE. "found in " some_data:$QUOTE.;
   end;
datalines;
Smith, BOB A.
ROBERT Allen Smith
Smithe, Cindy
103 Pennsylvania Ave. NW, Washington, DC 20216
508 First St. NW, Washington, DC 20001
650 1st St NE, Washington, DC 20002
3000 K Street NW, Washington, DC 20007
1560 Wilson Blvd, Arlington, VA 22209
1-800-123-4567
1(800) 789-1234
;
run;
```

1.6.5 Substitution Operator

While the substitution operator s// is not technically an option, it belongs here if only because it truly stands apart from the other items discussed in this section. Although the substitution operation is similar in appearance to the other options, it fundamentally changes the RegEx activity from a matching operation to a match-and-replace operation. Placing "s" in front of the surrounding slashes (//) signifies that the pattern is being used to replace the text being matched and insert the accompanying replacement text. This operator is another peek at additional functionality that is explored in the next chapter with SAS functions. Once a pattern is matched, we can then do a variety of things with that information. A great analogy for how this works in practice is the find-and-replace functionality provided by many word processing applications—except this is much more powerful. Also, notice that there is a third slash in the examples below (in the middle of the patterns). That additional slash denotes where the matching portion of the RegEx ends and the replacement portion begins. And notice something important in the last example: *everything is a string literal*. That's right, all the characters that occur between the second and third slash are treated as just characters. Table 1.39 provides some examples, but we will cover this in detail in the next chapter, where we also discuss how to insert more than just character strings.

Table 1.39: Examples Using s//

Usage	Matches	Replaces with
s/Stop/Go/	"Stop"	"Go"
s/Sing/Read/	"Sing"	"Read"
s/1\s?\(800\)\s?-\s?/1-800-/	"1 (800) - " …	"1-800-"

Note: This is a more advanced function that our test code is not set up to handle. You'll just need to accept it as true until we use it with some SAS code in the next chapter.

1.7 Zero-width Metacharacters

Zero-width characters, often called positional characters, are not matched in isolation because they do not have a width. They are used as an additional piece of information for making a proper pattern match. There are numerous examples for how these zero-width characters can be used. For instance, perhaps you want to match a particular word, but only if it occurs at the beginning of a line.

1.7.1 Start of Line

The metacharacter ^ matches the beginning of a line or string. Depending on the text that we are processing, we might know a priori that a new line signifies something specific. For example, we might be looking for the beginning of a new paragraph, which could be denoted by a new line in combination with a capital letter and no preceding white space. Or we might need to be prepared to match an address that includes a new line for the city, state, and zip. Table 1.40 provides examples.

Table 1.40: Example Using ^

Usage	Matches
/^Washington, DC 20007/	“ Washington, DC 20007”
/^\w+\b/	The first word in a string.

Note: This metacharacter is often used as the logical NOT symbol, including within the character class metacharacters discussed in Section 1.3 and in SAS code. So be careful not to get confused in its usage when shifting between contexts.

1.7.2 End of Line

The metacharacter $ matches the end of a line or string. There are numerous situations in which this might become relevant, similar to the reasons for the ^ metacharacter. Table 1.41 provides examples.

Table 1.41: Example Using $

Usage	Matches
/3000 K Street NW,$/	“3000 K Street NW, ”
/\$\d+?\.\d{2}\s*?$/	“$150.52 ”

1.7.3 Word Boundary

The metacharacter \b matches a word boundary. The \b RegEx assertion metacharacter is zero-width because it actually represents the invisible gap between two characters, with a \w character on one side and \W on the other. Therefore, when you use this metacharacter, you won't generate matches that contain the associated non-word character. Table 1.42 provides examples.

Table 1.42: Example Using \b

Usage	Matches
/Street\b/	"Street" from the substrings, "Street," "Street " … But does NOT match from the substring "Streets" etc.
/\b8\d{2}\b/	"800" "888" … from the substrings "(800)" "-888-" … But does NOT match from the substrings "18002" …
/\b\U[a-z]+\E\b/	Words in all caps. Without the second \b, the output would also include single capitalized letters from the front of a word.

1.7.4 Non-word Boundary

The metacharacter \B matches a non-word boundary (i.e., anywhere \b does not match). This is especially useful for matching root words or substrings without including the surrounding pieces of information. Table 1.43 provides examples.

Table 1.43: Examples Using \B

Usage	Matches
/read\B/	"read" from the substrings, "reads" "reading" "reader" … But does NOT match from the substring "read"
/\Bun\b/	"un" from the substrings, "fun " "rerun." "gun," … But does NOT match from the substring "un"
/\b[a-zA-Z]{3,}\b/	Any word longer than three letters.

1.7.5 String Start

The metacharacter \A matches the beginning of a string. Similar to the word boundary metacharacter (\b), \A occurs between two character cells. It also denotes when a string value occurs to its right with nothing to its left. In the context of data lines (as in our test code for this chapter), that situation occurs at the beginning of each line.

However, suppose we had a more complex task such as stitching together multiple strings of extracted text (stored in SAS variables). In this context, \A could be a key to determining in what order to place or sort them. However, for our test code, the \A matches only on the beginning of each data line, since each line is identified as the beginning of the string. So, this is another one that you have to approach with a little bit of faith until we start doing some more interesting tasks in the next chapter. Table 1.44 provides examples.

Table 1.44: Examples Using \A

Usage	Matches
/\A\w*?\s/	The first word of a line. In the case of our test code, it matches: “ROBERT ” from line 2; “103 ” from line 4; “508 ” from line 5; “650 ” from line 6; “3000 ” from line 7; and “1560 ” from line 8.

1.8 Summary

We have explored a variety of interesting new concepts in this chapter, and I’ve been doing my utmost to make them tangible along the way. Hopefully, you are now ready to tackle the challenge of implementing these concepts in SAS code in the coming chapters. Following are some takeaways that you should keep in mind for the coming pages and beyond.

Flexibility

It should have become clear through reading this chapter that there are many ways to accomplish the same task, making few of them truly right or wrong. You have to decide the most efficient and effective approach for accomplishing your goals to determine what is best for a given situation.

Scratching the Surface

We have only begun to scratch the surface of what RegEx can do. The information that you have learned thus far is a solid foundation upon which you can develop sophisticated functionality.

Start Small

As we have explored a variety of RegEx capabilities throughout this chapter, it is easy to become overwhelmed with attempting to do too much at once. As with anything, it is best to start small by experimenting with simple patterns and iteratively evolve them. And remember that leveraging just a few of the elements that we have covered can have a tremendous impact on the processing and analysis of textual information.

1 Wikipedia contributors, "Regular expression," *Wikipedia, The Free Encyclopedia*, https://en.wikipedia.org/w/index.php?title=Regular_expression&oldid=857059914 (accessed August 29, 2018).

2 For more information on the version of Perl being used, refer to the artistic license statement on the SAS support site here: http://support.sas.com/rnd/base/datastep/perl_regexp/regexp.compliance.html

3 SAS Institute Inc. "Tables of Perl Regular Expression (PRX) Metacharacters," SAS 9.4 Functions and CALL Routines: Reference, Fifth Edition, http://support.sas.com/documentation/cdl/en/. lefunctionsref/67239/HTML/default/viewer.htm#p0s9ilagexmjl8n1u7e1t1jfnzlk.htm (accessed August 29, 2018).

4 "perlre," *Perl Programming Documentation,* http://perldoc.perl.org/perlre.html (accessed August 29, 2018).

5 SEC, "SEC Administrative Proceedings for 2009," *U.S. Securities and Exchange Commission*, http://www.sec.gov/open/datasets/administrative_proceedings_2009.xml (accessed August 29, 2018).

6 Octal is a number system that uses base-8 instead of base-10. This system has only numbers 0–7 represented. Some old microcontrollers and microprocessors used this encoding, but it is extremely rare today.

7 Hexadecimal is a number system that uses base-16 instead of base-10. The possible values go from "0" to "F" in a single character position (where A=10, B=11, …, F=15).