ADVANCED PHP STRINGS

Text analysis, generation, and parsing via. Laravel

John Koster

Advanced PHP Strings

Text analysis, generation, and parsing via.

Laravel

Johnathon Koster

This book is for sale at http://leanpub.com/advanced-php-strings

This version was published on 2022-10-06

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.

Carla and Eric, for whom "actions speak louder than words," applies more than most.

Contents

Introduction

We will cover many topics throughout this book, from working with Unicode character sets and the Laravel helper methods to parsing and splitting complex strings. This book has been arranged such that each topic builds off the concepts that appear before it within the book, but there is no particular recommended order for reading this book.

You are encouraged to find a topic that interests you and jump to that section. When doing so, you may find that the discussion references previous items discussed in the book. When this happens, you may continue reading through your selected section. If the examples become challenging, you may wish to return to earlier parts of the book that help build foundational knowledge for your chosen area.

The topics listed below do not include everything each chapter provides and are intended to help give a general idea of what each chapter contains beyond its title. Exploration of this book's content is encouraged!

Chapter 1: What are Strings: This chapter covers Unicode, UTF-8, the different types of PHP strings, and techniques to iterate multi-byte PHP strings.

Chapter 2: Fluent Strings: This chapter introduces Laravel's fluent string helper class, which is used extensively throughout the book.

Chapter 3: The Formatting Helper Methods: This chapter covers the Laravel string helper methods that are typically used to help format or transform an existing string in some way. Examples include title casing and converting titles to URL slugs.

Chapter 4: The Logical Helper Methods: The logical helper methods assist in making decisions around our strings. Topics include determining if an input string contains one or more other strings and checking if our text starts or ends with particular values.

Chapter 5: The Construction Helper Methods: The construction helper methods allow us to generate new strings from existing input or to construct pseudo-random strings. Topics include replacing substrings using various techniques and ensuring input always ends with a given character sequence.

Chapter 6: The Extraction Helper Methods: The extraction helper methods are a set of utility methods that allow us to retrieve parts of existing strings using various techniques. Examples include converting input into a list of words and retrieving the content between two search strings.

Chapter 7: Padding Strings: This chapter thoroughly discusses the concept of string padding. Topics include simple string padding, advanced methods for number formatting, and more.

Chapter 8: String Translations; Singular and Plural Word Forms: This chapter discusses the various features Laravel provides to help work with, manage, and manipulate translation strings.

Chapter 9: Extending Laravel Strings with Macros and Mixins: This chapter provides an overview of how to extend Laravel's string helper class and the fluent string class. Extending the Laravel helpers is used frequently throughout this book.

Chapter 10: Lines and Words: This chapter focuses on extracting line and column information from the input text and various techniques to extract and analyze words.

Chapter 11: More Lines and Words: This chapter continues the discussion surrounding lines and words and offers techniques for retrieving adjacent words and characters from arbitrary positions. Additional topics include calculating an estimated reading time and generating initialisms from input text.

Chapter 12: Applied Techniques: Writing a Gherkin Parser: This in-depth chapter utilizes techniques discussed in previous chapters to build a Gherkin parser from start to finish.

Chapter 13: Markov Chains and Text Generation: This chapter explores generating randomized text using Markov Chains. The discussion includes creating our training methods and implementing a weighted selecting algorithm to help generate our text.

Chapter 14: Fixed With Data Parsing: This chapter discusses various techniques for parsing fixed-width data commonly found in exports from financial institutions.

Chapter 15: Splitting Strings: The discussions in this chapter surround advanced techniques and methods for splitting complicated input text using non-trivial rules and requirements. The discussion concludes with building a general-purpose, reusable cursor system to make extracting information from strings easier.

Chapter 16: Applied Techniques: A Blade Directive Validator: This chapter applies most of the techniques covered in the book to implement a simple Blade directive parser and validation system. Validation logic includes detecting when directives are missing required arguments, have inconsistent spacing, or when errant newlines may cause compiler errors.

Chapter 17: Working with HTML: This chapter explores techniques for reading and manipulating HTML documents using Symfony framework components. Methods are also examined for parsing and analyzing semantically invalid HTML documents, such as those that contain embedded templating languages.

Chapter 18: Regular Expressions: The discussion of this chapter focuses on the most common regular expression features encountered when working on various code bases. Topics include the Laravel pattern-matching methods, negative look-behinds, and a regular expression language primer.

PHP Version

Throughout this book, we will explore many different concepts and ideas through many code examples. PHP 8 was used throughout the writing process, but the code examples can be adapted to any modern PHP version.

As a consequence of deciding to use PHP 8, there was no consideration made for the overloading of the standard string functions using PHP's function overloading feature, as this was removed in PHP version 8.0.0[1].

Laravel Version

The most recent version of Laravel, version 9.26.1, was used when writing the examples we will find throughout the book. Because of the frequency of updates to the Laravel framework, there may be delays in seeing newly added helper methods covered throughout the various chapters.

[1]For more information about PHP's removed function overloading feature, you may consult the official documentation at: https://www.php.net/manual/en/mbstring.overload.php.

Although we will assume the most recent version of Laravel for our code examples and exercises, the method signatures will outline any difference between Laravel 8 and Laravel 9 to help upgrade processes.

When we introduce new helper methods throughout the book, we will see an overview of the signature of the method. Where we would typically see a description of the method in typical PHP source code, we will see which versions of the Laravel framework support the helper method.

For example, the `lcfirst` helper method was introduced in Laravel 9 and will have the following method signature:

```php
<?php

/**
 * Version: Laravel 9
 *
 * @return string
 */
public static function lcfirst(
    string $string
);
```

In contrast, the `lower` helper method appears in Laravel versions 8 and 9, with no differences in the method signature between the two versions:

```php
<?php

/**
 * Versions: Laravel 8, 9
 *
 * @return string
 */
public static function lower(
    string $value
);
```

If a helper method is present in both Laravel 8 and 9 but has a different signature between the two versions, the method signature will list both so we can more easily see the differences. An example of this is the `wordCount` method signature:

```php
<?php

// Version: Laravel 8
public static function wordCount(
    string $string
);

// Version: Laravel 9
public static function wordCount(
         string $string,
    string|null $characters = null
);
```

Symbols Used In This Book

We will encounter a small number of icons throughout this book. These icons call out additional information and potential pitfalls or direct you to other time-saving resources.

Look for this icon in the text to learn about any appendices that provide additional reference material or context for a given section.

This icon alerts you to additional information that you may find interesting or provides further context around ideas to consider when implementing them in your projects.

This icon calls out information that may be technically correct for a given situation. Still, there may be issues that lead to common bugs. It will often direct you to a different chapter or section that addresses the issue.

Hic sunt dracones

1. What are Strings

For a book that aims to explore PHP strings through the lens of the Laravel string helper functions, it seems obvious to start with the most basic question first: what even *are* strings? The first answer that might come to people is that strings are simply a list of characters. While, on the surface, this seems to satisfy the question, we are only three sentences into this book and have already hit our first set of problems, which leads to even more questions.

The first such question might be: what is a character? When we look at the text on a screen or in printed media, we naturally assume that each visually distinct unit of a piece of writing is a character. For example, if you grew up speaking and writing English, it is second nature to look at the following text and be able to see that it contains sixteen distinct characters quickly:

Figure 1.1

```
1 Hello wilderness
```

But what about the following?

Figure 1.2

```
1 你好荒野
```

Suppose you are reading a digital copy of this book. In that case, it is possible to highlight each character in that text, and you would ascertain that the previous text contains four characters. Let us make doubly sure of this and use PHP's `strlen` function to count the characters in this string:

Figure 1.3

```php
<?php

// Returns 12
strlen('你好荒野');
```

The `strlen` function returns twelve. That's interesting! Is it broken? Let's use it to count the number of characters in our English text:

Figure 1.4

```php
<?php

// Returns 16
strlen('Hello wilderness');
```

This time the function returns sixteen, which is the correct number of characters in our text. What's going on here? To answer this question, we should take a step back and think about how we store and represent text in computer memory.

1.1 A Brief History of Character Encodings

Suppose we must develop a compact way of sending messages between two groups, one that reduces the total amount of data sent but does not lose any information. Let's say we only need to concern ourselves with words, not numbers or special characters. We could analyze previous communication between the two parties, find the frequent words and phrases, and represent them as numbers or symbols. When we want to send a new message, we will replace those words and phrases with their corresponding number or symbol. When the recipient of the message goes to read the message, they would consult the table of numbers and symbols and reverse the process to get back the original message. We would have developed an encoding scheme: a way to represent and interpret data.

When we look at the history of text data and character encodings within computer systems, it does not take long for the name ASCII, an abbreviation of the American

Standard Code for Information Interchange, to come up, which has its roots in telegraph codes developed at Bell.

In the early days of computing, it was widespread for each company or organization producing a computer system to go through the same exercise of creating a unique way of storing text: to make a unique character encoding system. These systems were rarely compatible, and many parties, with the support and direction of the then American Standards Association, undertook work throughout the mid to late 1990s to produce a standard way of encoding text.

Much of this work was not widely adopted until March 11, 1968, when U.S. President Lyndon B. Johnson mandated that ASCII become a U.S. federal standard. In addition to promoting ASCII to a federal standard, the President also made it a requirement that all computers and related equipment purchased by the U.S. government starting July 1, 1969, be ASCII-compatible. Fast forward from these early days of the development of the standard (although there were many revisions to it in the following years) to 1981, when IBM used it to encode text with their first personal computer.

We will skip a lot more history and nuance, but we have enough context to proceed. Let's now work to understand what all of this has to do with the wrong results of the strlen function when attempting to count the number of characters in our piece of Traditional Chinese text. The first fact to know is that ASCII was initially devised as a 7-bit standard and provided an encoding for 128 characters. The nice thing about this is that each character is easily represented by a single byte within a computer. Or, put another way, each character is mapped to an integer between 0 and 255, the maximum value we can represent with a single byte.

If we were to convert our English string into a byte array using:

Figure 1.5

```
1  <?php
2
3  $bytes = unpack('C*', 'Hello wilderness');
```

We would get the following array of integers:

Figure 1.6

```
array:16 [
     1 => 72
     2 => 101
     3 => 108
     4 => 108
     5 => 111
     6 => 32
     7 => 119
     8 => 105
     9 => 108
    10 => 100
    11 => 101
    12 => 114
    13 => 110
    14 => 101
    15 => 115
    16 => 115
]
```

If we were to look up the ASCII table for those values, we would find that those numbers code for each of the characters in our original string. Let's now look at the results of the following:

Figure 1.7

```
<?php

$bytes = unpack('C*', '你好荒野');
```

Which produces:

Figure 1.8

```
array:12 [
     1 => 228
     2 => 189
     3 => 160
     4 => 229
     5 => 165
     6 => 189
     7 => 232
     8 => 141
     9 => 146
    10 => 233
    11 => 135
    12 => 142
]
```

If you notice, our Traditional Chinese string is a byte array containing twelve distinct values, which leads us to the next piece: the `strlen` function is counting the *number of bytes*.

For English characters in ASCII, counting the number of bytes is equivalent to counting the number of characters in the string: all English characters can be represented by a single byte. So how does this work with the Traditional Chinese text? Let's start by breaking down each character and checking what the byte array it produces is:

Figure 1.9

```
// unpack('C*', '你');
array:3 [
    1 => 228
    2 => 189
    3 => 160
]

// unpack('C*', '好');
array:3 [
```

```
10      1 => 229
11      2 => 165
12      3 => 189
13  ]
14
15  // unpack('C*', '荒');
16  array:3 [
17      1 => 232
18      2 => 141
19      3 => 146
20  ]
21
22  // unpack('C*', '野');
23  array:3 [
24      1 => 233
25      2 => 135
26      3 => 142
27  ]
```

As you can see, each of those characters is composed of *multiple* bytes, which is where the name multibyte comes from in the context of PHP's multibyte string functions. One of the multibyte string functions is the `mb_strlen` function, which is the multibyte version of the strlen function. Using this function on our Traditional Chinese text returns the value four, which matches what we visually see:

Figure 1.10

```
1  <?php
2
3  // Returns 4
4  mb_strlen('你好荒野');
```

1.2 Unicode and UTF-8

In the previous section, we looked at the underlying bytes leading to strange results when using the `strlen` function. We also discussed a bit of the history of character

encodings in general, but one thing we did not answer was: what was the character encoding used when converting our strings to byte arrays? When we looked at those byte arrays, we could see the representation of those characters. Still, without knowing the character encoding, all of those values are meaningless since we would not know how to interpret them. The short answer we could give here is, "well, it's Unicode," but that only gets us so far.

Unicode, like ASCII, is a character set that provides a mapping of characters to an integer (Unicode refers to these associations as code points). However, unlike ASCII, Unicode does not dictate how those values are stored or transferred (remember, in ASCII, characters are mapped to and persisted as single bytes).

Let's take a look at our Traditional Chinese text again, but with the Unicode code points:

Table 1.1 Unicode Code Points Example

Character	Code Point	Name
你	U+4F60	CJK Unified Ideograph-4F60
好	U+597D	CJK Unified Ideograph-597D
荒	U+8352	CJK Unified Ideograph-8352
野	U+91CE	CJK Unified Ideograph-91CE

We can identify our four characters within the Unicode character set by referring to their code point. The code point maps a number to a character or position within the Unicode character set, and Unicode code points begin with the "U+" prefix followed by a hexadecimal number. To help build our understanding of how code points relate to the Unicode character set, we can look at a feature introduced in PHP 7: Unicode code point escape syntax.

From the PHP manual, the Unicode code point escape syntax "takes a Unicode code point in hexadecimal form, and outputs that codepoint in UTF-8". Earlier, we discussed how the code point is the "U+" prefix followed by a hexadecimal number. Starting with PHP 7, we can embed those same characters by referring to their Unicode code point in hexadecimal form, which we can get from the "Unicode Code Points Example" table:

Figure 1.11

```php
<?php

// Outputs " 你好荒野"
echo "\u{4F60}\u{597D}\u{8352}\u{91CE}";
```

Apart from language-specific escape sequence characters, PHP's code point escape syntax reads almost identical to the Unicode code point itself. However, one essential phrase in the PHP manual's explanation was "and outputs that codepoint in UTF-8." What is UTF-8, and why does it keep appearing in this discussion about Unicode, characters, and text? If Unicode describes the character set, UTF-8 is one of three *Unicode transformation formats*.

We, as developers, will most commonly encounter UTF-8 within the context of web applications, websites, or other Internet-connected applications. With UTF-8, we can encode Unicode using one, two, three, or four *eight*-bit bytes.

To explore this, let's have fun with the "你" character. To get started, we will take the hex component of the code point and convert it to a decimal number using PHP's `hexdec` function:

Figure 1.12

```php
<?php

// Returns 20320
hexdec('4F60');
```

Converting the hexadecimal component to a decimal produces the result 20320. If we were to look up the HTML code for the 你 character, we would see that it is:

Figure 1.13

```
&#20320;
```

Notice how the character's HTML and decimal versions are almost the same? Pretty interesting! We will now take it a step further and work to take our decimal number

and convert it to binary, and eventually arrive at the same array of integers we saw in the previous section.

We will start this process using PHP's `decbin` function to convert our decimal number into binary. Doing this returns the binary value `100111101100000`:

Figure 1.14

```
<?php

// Returns "100111101100000":
decbin(20320);
```

To get this value encoded using UTF-8, we will have to look at the rules for encoding values.

The rules we will look at are adapted from RFC 3629, which provides ranges of numbers and their sequence of an eight-bit byte, or octet "templates."

Octet is one of those words that can be intimidating at first: it just feels complicated and advanced. In reality, it often just refers to a group of eight things. In our case, it refers to a byte of precisely eight bits: eight ones or zeroes. Most modern developers will have only ever encountered bytes eight bits in length, but this has not always been the case historically. We need not look further back than the discussion about ASCII, which stored characters in seven-bit bytes.

With UTF-8, we will encode characters in an octet *sequence*: a series of either one, two, three, or four octets. To encode a character with UTF-8, we need to know its numeric value or Unicode code point. The size of this number in bytes is what we will use to determine which UTF-8 octet sequence to use. The following table lists the numeric ranges and the octet template to use:

Table 1.2 UTF-8 Octet Sequence Ranges

Decimal Range	Hex Range	Octet Sequence
0 - 127	0000 0000 - 0000 007F	0xxxxxxx
128 - 2,047	0000 0080 - 0000 07FF	110xxxxx 10xxxxxx

Table 1.2 UTF-8 Octet Sequence Ranges

Decimal Range	Hex Range	Octet Sequence
2,048 - 65,535	0000 0800 - 0000 FFFF	1110xxxx 10xxxxxx 10xxxxxx
65,536 - 1,114,111	0001 0000 - 0010 FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

We will use the decimal values to compare in the table, but most implementations will use the hex values. Earlier, we found that the decimal value of the code point for the "你" character was 20320. Looking at our table, we can see that we will need to use the following octet sequence template:

Figure 1.15

```
1110xxxx 10xxxxxx 10xxxxxx
```

We will also need our character's binary data now that we have our template. Luckily, the value `100111101100000` we came up with earlier is just what we need for this next step. We now need to replace the x's in the octet template with the binary data.

We do this by starting at the template's right-hand side and replacing the last x with the last bit in the binary data. After this, we continue to move toward the left until we run out of data. Once complete, we should end up with the following result:

Figure 1.16

```
1110x100 10111101 10100000
```

As you can see, we have one x left over in the result. Anytime we have left-over placeholders, we simply replace them with 0's to arrive at (**Figure 1.18** visually represents where we replace the bits in our sequence template):

Figure 1.17

```
11100100 10111101 10100000
```

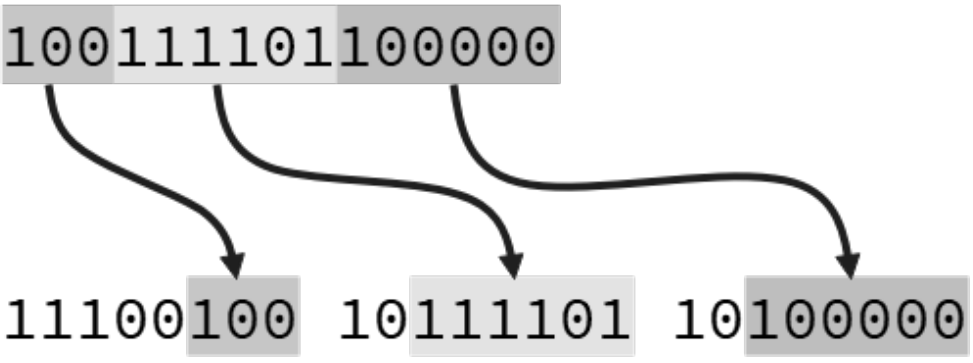

Figure 1.18 Encoding UTF-8 Values

If we were to look up the UTF-8 binary value for "你," we would find that it matches the value we produced manually. At the beginning of this chapter, we looked at how we could use the `unpack` function to retrieve a character's constituent values. As a refresher, here are that function's results for "你":

Figure 1.19

```
// unpack('C*', ' 你');
array:3 [
    1 => 228
   2 => 189
    3 => 160
]
```

To bring things full circle, we can now use PHP's `bindec` function to convert the binary values we arrived at manually to their decimal value:

Figure 1.20

```php
<?php

// Returns 228
bindec('11100100');

// Returns 189
bindec('10111101');

// Returns 160
bindec('10100000');
```

1.3 Implementing Upper and Lowercase Functions

As stated earlier, one of the benefits of UTF-8 is that it preserves the ASCII values[1] and their relative positions. We can explore working more closely with strings by reimplementing the upper and lowercase functions. We will only focus on the ASCII character range to keep things simple.

Table 1.3 ASCII Alpha Characters

ASCII Code	Symbol	ASCII Code	Symbol
65	A	97	a
66	B	98	b
67	C	99	c
68	D	100	d
69	E	101	e

[1]It is important to note that UTF-8 includes only the first 128 ASCII characters (0-127) and does not contain the extended ASCII character set.

Table 1.3 ASCII Alpha Characters

ASCII Code	Symbol	ASCII Code	Symbol
70	F	102	f
71	G	103	g
72	H	104	h
73	I	105	i
…	…	…	…
87	W	119	w
88	X	120	x
89	Y	121	y
90	Z	122	z

From the table, we can notice an interesting pattern with the ASCII characters: all of the lowercase character's decimal values are thirty-two greater than their uppercased variant. This ASCII code distribution means that if we want to convert all uppercase characters to lowercase, we need to add thirty-two to its value.

Figure 1.21 uppercaseString Implementation

```
<?php

function uppercaseString($input) {
    $newString = '';

    for ($i = 0; $i < strlen($input); $i++) {
      $firstByte = ord($input[$i]);

      if ($firstByte >= 97 && $firstByte <= 122) {
        $newString .= chr($firstByte - 32);
      } else {
        $newString .= chr($firstByte);
      }
```

```
    }

    return $newString;
}
```

Figure 1.22 lowercaseString Implementation

```
<?php

function lowercaseString($input) {
    $newString = '';

    for ($i = 0; $i < strlen($input); $i++) {
      $firstByte = ord($input[$i]);

      if ($firstByte >= 65 && $firstByte <= 90) {
        $newString .= chr($firstByte + 32);
      } else {
        $newString .= chr($firstByte);
      }
    }

    return $newString;
}
```

Our two functions can be used like so:

Figure 1.23

```
<?php

// Returns "HELLO, WORLD!"
uppercaseString('hello, world!');

// Returns "hello, there!"
lowercaseString('HELLO, there!');
```

We saw two new PHP functions in the previous code samples: `ord` and `chr`.

The `ord` function will return the first byte of a string value as a value between 0 and 255. For our purposes here, this will effectively tell us the character's position within the ASCII range. The `chr` function does the opposite: it will take a value between 0 and 255 and convert it to its ASCII character.

If we wanted a little bit of fun, we could combine these functions to swap the casing of each letter in the input string. Because ASCII characters are thirty-two bytes away from their counterparts, we can use the bitwise XOR operator to swap the positions of our letters:

Figure 1.24 swapCase Implementation

```
<?php

function swapCase($input) {
    $newString = '';

    for ($i = 0; $i < strlen($input); $i++) {
      $firstByte = ord($input[$i]);

      if ($firstByte >= 65 && $firstByte <= 90 ||
          $firstByte >= 97 && $firstByte <= 122) {
        $newString .= chr($firstByte ^ 32);
      } else {
        $newString .= chr($firstByte);
      }
```

```
    }

    return $newString;
}
```

Our `swapCase` function would return the following results given the sample input:

Figure 1.25

```
<?php

// Returns "hElLo, WoRlD!"
swapCase('HeLlO, wOrLd!');

// Returns "hellO, wOrld!"
swapCase('HELLo, WoRLD!');
```

Our simple implementations also use a loop to iterate the characters of our string. With ASCII being one-byte-per-character, this is completely fine. However, as you may have guessed, this becomes a bit more challenging once we move into multi-byte characters.

1.4 Iterating Multibyte Strings

UTF-8 String Iterator Implementation

Appendix A: UTF-8 String Iterator Implementation contains the full implementation for the class we will develop throughout this chapter. You are encouraged to work through the sections of this chapter, but if you only want the code feel free to grab it from there.

In the previous section, we implemented several string functions that worked by iterating a string's characters. Treating each byte as a single character is fine when working with ASCII, but as soon as we need to work with multibyte characters, things become a little more complicated. As an example, if we attempted to iterate "這可以" using the following:

Figure 1.26

```
<?php

$string = '這可以';

$output = '';

for ($i = 0; $i < strlen($string); $i++) {
    $output .= $i . ': '. $string[$i];
}
```

Our output string would appear corrupted:

Figure 1.27

```
0: Ú1: Ç2: Ö3: Õ4: Å5: »6: õ7: ╗8: Ñ
```

We can attempt to fix this by replacing the `strlen` call with `mb_strlen`:

Figure 1.28

```
<?php

$string = '這可以';

$output = '';

for ($i = 0; $i < mb_strlen($string); $i++) {
    $output .= $i . ': '. $string[$i];
}
```

However, this time we would receive the following output, which also does not look correct:

Figure 1.29

```
0: é1: €2: ™
```

A quick fix for this would be to use the `mb_str_split` function to break our input string into an array of characters and iterate that:

Figure 1.30

```
<?php

$output = '';

$chars = mb_str_split($string);

for ($i = 0; $i < count($chars); $i++) {
    $output .= $i . ': '. $chars[$i];
}
```

Which produces the result:

Figure 1.31

```
0: 這1: 可2: 以
```

However, as the input size grows, this method can lead to substantial memory consumption. Depending on the use case, this may be fine. Other applications, such as those that parse arbitrary user input on every request, may not be able to get away with such high memory consumption, particularly if the parsing function is being invoked many times across multiple simultaneous requests.

Before diving directly into iterating multibyte strings, we will take a quick detour and see what it would look like if we were to implement the mb_strlen function for UTF-8 strings ourselves.

In our discussion about Unicode and UTF-8, we saw a table listing the different ranges of values and their octet sequences used to encode UTF-8 strings. We have also encountered the `ord` function while implementing various functions, which returns

the first byte of a string as a value between 0 and 255. These two pieces of information will become very useful when we implement our length function.

As the first step in implementing our length function, let's take another look at the UTF-8 octet sequence table, but this time let's figure out what the maximum value of the first byte could be:

Table 1.4 UTF-8 Octet Sequence First Byte Maximum Values

Sequence	Max (Binary)	Max (Decimal)	Byte Length
0xxxxxxx	01111111	127	1
110xxxxx 10xxxxxx	11011111	223	2
1110xxxx 10xxxxxx 10xxxxxx	11101111	239	3
11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	11110111	247	4

We know that we can use the `strlen` function to get the total number of *bytes* in a string and the `ord` function to return a numeric value of a byte. Using this, we can iterate a string's bytes and determine its UTF-8 sequence. For example, using the ord function on "A" would return 16, meaning it would use the first sequence, and we can expect one byte. Calling ord with "這" would yield 233, which falls within the range of the third sequence. Because of this, we can expect three bytes.

With all of this information in mind, we can implement our length function like so:

Figure 1.32

```
<?php

function utf8_strlen($input) {
  $charCount = 0;

  for ($i = 0; $i < strlen($input); $i++) {
      $charCount += 1;
      $byte = ord($input[$i]);

      if ($byte <= 127) {
```

```
            continue;
        } else if ($byte <= 223) {
            $i += 1;
        } else if ($byte <= 239) {
            $i += 2;
        } else {
            $i += 3;
        }
    }

    return $charCount;
}
```

We can compare the output of our function to the results of the `mb_strlen` function:

Figure 1.33

```
<?php

// Returns 16
utf8_strlen('這可以abcd!! 這可以abcd');

// Returns 16
mb_strlen('這可以 abcd!! 這可以 abcd');
```

Our implementation checks the byte value against our known ranges and uses that information to skip ahead in the string. You will notice that each time we skip a number of bytes in the string, we always skip one less than what we see in the table. We do this because the for loop will add one to our counter variable after each iteration.

Throughout the rest of this chapter, we will work to implement the following class that we can use to iterate UTF-8 strings:

Figure 1.34

```
<?php

$iterator = new Utf8StringIterator('這可以-this is fine!');

foreach ($iterator as $char) {
    echo $char."\n";
}
```

After our implementation is complete, the above will produce the following output:

Figure 1.35

```
這
可
以
-
t
h
i
s

i
s

f
i
n
e
!
```

We will use all the techniques we've discussed so far, with the significant difference being the implementation details of various PHP interfaces. We will start with the following empty class and work to fill in all of the methods:

2. Lines and Words

Throughout this chapter, we will explore the concepts of lines and words. Both terms are ubiquitous: we see them everywhere but probably do not give them much thought. We will start with lines.

Back in "Chapter 9: Extending Laravel Strings with Macros and Mixins" we put together a quick example macro that would break a corpus into an array of individual lines:

Figure 10.1

```
<?php

use Illuminate\Support\Str;

Str::macro('lineSplit', function ($value) {
    return explode("\n", $value);
});
```

When we did this in that chapter, we did note that this implementation had some issues. Namely, it does not handle the line ending types we often encounter. Our current line splitting implementation only accounts for the "\n" (line feed) character, which will work well with text input or documents from UNIX or modern macOS systems. However, there are others.

For example, older macOS systems would use the "\r" (carriage return), and Windows systems use the "\r\n" sequence (carriage return followed by a line feed). These characters are interesting in their own right. Most developers will have interacted with completely digital systems and may not know that each of these characters has historical use and significance.

Back in "Chapter 1: What are Strings?" we discussed ASCII, Unicode, and character encodings. One of the interesting things about character sets is that they are often

subdivided into at least two groups: control characters and printable characters. The various newline characters fall into the former group of control characters. Control characters cause a physical system to do something: they *control it*. Let's take a moment and think about a typewriter.

When we type characters on a typewriter, a hammer is flung through the air, hits an ink strip, and stamps the symbol on our page. Each time this happens, the carriage, the apparatus that advances the page as we type and moves it up and down, moves to the left slightly. If we do this enough, the carriage will eventually get as far to the left as it can and will not be able to advance any further; a satisfying bell ring often accompanies this. To start a new line, we hit the return key, which causes the wheel holding the paper to rotate slightly. The carriage returns to its original position, allowing us to start typing at the start of the new line.

Pressing the return key initiates two actions: the paper feeds through the roller, and the carriage *returns* to its starting position.

Now that we have a bit of context surrounding the different line ends we commonly see, we can ask ourselves another question: are there other characters we need to be aware of? There are, in fact, many different control characters that have historically been used that cause a new line to be inserted or indicate some vertical separation. **Table 10.1** provides a list of the possible newline character sequences we should be aware of:

Table 10.1 Possible Newline Characters

Character Sequence	RegEx Pattern	Unicode Code Point
Line Feed (LF)	/\n/	U+000A
Carriage Return (CR)	/\r/	U+000D
CR+LF	/\r\n/	
LF+CR	/\n\r/	
Form Feed (FF)	/\x{000C}/	U+000C
Next Line (NEL)	/\x{0085}/	U+0085
Line Separator (LS)	/\x{2028}/	U+2028

Table 10.1 Possible Newline Characters

Character Sequence	RegEx Pattern	Unicode Code Point
Paragraph Separator (PS)	/\x{2029}/	U+2029
Vertical Tab (VT)	/\x{000B}/	U+000B

Each of the different characters in **Table 10.1** has its use. For example, the vertical tab character instructs a printer or display to advance a single line while preserving the horizontal spacing. For our purposes, however, we will only focus on the line feed and carriage return characters.

Using the regular expression (regex) patterns from **Table 10.1**, we can construct a regex pattern that we can use with PHP's `preg_split` function. Our regex will accept either the carriage return followed by the line feed sequence *or* the line feed character *or* the carriage return character:

Figure 10.2 regex Pattern to Match Line Endings

```
/(\r\n|\n|\r)/
```

We can now update the macro we created in the previous chapter:

Figure 10.3

```
<?php

use Illuminate\Support\Str;

Str::macro('lineSplit', function ($value) {
    return preg_split("/(\r\n|\n|\r)/u", $value);
});
```

After this change, our `lineSplit` method will now be able to handle the most common of the new line characters. We also have the opportunity to create another helper method: one to *normalize* the line endings that appear within a piece of text:

Figure 10.4

```php
<?php

use Illuminate\Support\Str;

Str::macro('normalizeEol', function ($value, $to = PHP_EOL) {
    return preg_replace("/(\r\n|\n|\r)/u", $to, $value);
});
```

We can use our new `normalizeEol` method like so to convert line endings:

Figure 10.5

```php
<?php

use Illuminate\Support\Str;

$string = "line one\nline two\r\nline three\rline four";

// Returns "line one\nline two\nline three\nline four"
Str::normalizeEol($string, "\n");
```

An interesting aspect of the `normalizeEol` signature is the default value of the `$to` parameter: `PHP_EOL`. The `PHP_EOL` constant will return the line ending style for the platform that PHP is running on; for example, it will be set to "\r\n" on a Windows system.

We can also update our `lineSpli` fluent string macro and add a `normalizeEol` method as well:

Figure 10.6

```php
<?php

use Illuminate\Support\Stringable;

Stringable::macro('normalizeEol', function ($to = PHP_EOL) {
    return preg_replace("/(\r\n|\n|\r)/u", $to, $this->value);
});

Stringable::macro('lineSplit', function () {
    return collect(preg_split("/(\r\n|\n|\r)/u", $this->value));
});
```

Using our new fluent `lineSplit` method like so:

Figure 10.7

```php
<?php

str("line one\nline two\r\nline three\rline four")
    ->lineSplit;
```

Would return a new Collection containing the following elements:

Figure 10.8

```
[
    "line one",
    "line two",
    "line three",
    "line four",
]
```

2.1 Positions, Lines, and Columns

CharacterPositionLocator Implementation

Appendix B: CharacterPositionLocator Implementation contains the full implementation for the class we will develop throughout this section. You are encouraged to work through this section to help build an understanding of how each of the methods was implemented.

Over the years, I have worked on many text-related projects that have involved working with lines, positions, and columns. For example, when we look at a cursor's position with a text editor, we often see it displayed in lines and columns (e.g., Ln 7, Col 62). From our discussions in Chapter 1, we know that strings are represented in memory as a contiguous array of bytes. The question then becomes: how do we apply two-dimensional concepts on top of a unidimensional array of bytes?

Unpacking the following string value:

Figure 10.9

```
1  <?php
2
3  $text = "one\ntwo\nthree four\nfive";
4
5  unpack('C*', $text);
```

Returns the following list of bytes in decimal form:

Figure 10.10

```
array:23 [
     1 => 111
     2 => 110
     3 => 101
     4 => 10
     5 => 116
     6 => 119
     7 => 111
     8 => 10
     9 => 116
    10 => 104
    11 => 114
    12 => 101
    13 => 101
    14 => 32
    15 => 102
    16 => 111
    17 => 117
    18 => 114
    19 => 10
    20 => 102
    21 => 105
    22 => 118
    23 => 101
]
```

Looking at the results in **Figure 10.10**, we can glean a few clues as to what we can do to convert arbitrary string positions into their line and column numbers. For instance, we know that the decimal value 10 corresponds to the newline character. To get the current line number from a position within the string, we can count the number of times this character has appeared before the current position. The following code example demonstrates a simple way we could convert any string position into a line number:

Figure 10.11

```php
<?php

$text = "one\ntwo\nthree four\nfive";

function getLineNumber($value, $position) {
    return mb_substr_count(
        mb_substr($value, 0, $position),
        "\n"
    ) + 1;
}

// Get line number of the first "o".
// Returns 1
getLineNumber($text, 0);

// Get the line number of the first "t".
// Returns 2
getLineNumber($text, 4);
```

This initial implementation works by using the desired position and extracting a substring from the original string. We then count the number of newline characters that appear in the resulting string. The line count will be zero-based, so we add one to get a human-friendly line number. This technique works well on smaller strings but can become quite memory intensive, creating the substrings on huge strings. We will explore ways to address this later.

We can employ a similar strategy to determine the column number for any position in the string. The basic technique will be to get the position of the newline character that immediately precedes the desired position. Once we have both positions, the column number will be the difference between the two positions.

Figure 10.12

```php
<?php

$text = "one\ntwo\nthree four\nfive";

function getColumnNumber($value, $position) {
    $lastNlPosition = mb_strrpos(
        mb_substr($value, 0, $position),
        "\n"
    );


    if ($lastNlPosition === false) {
        return $position + 1;
    }

    return $position - $lastNlPosition;
}

// Returns 1
getColumnNumber($text, 0);

// Returns 3
getColumnNumber($text, 2);

// Returns 1
getColumnNumber($text, 8);

// Returns 8
getColumnNumber($text, 15);
```

Using the string iterator developed in Chapter 1, we can construct a loop to help better visualize the output of these two functions. The following code example:

Figure 10.13

```php
<?php

$text = "one\ntwo\nthree four\nfive";

$string = new Utf8StringIterator($text);

for ($i = 0; $i < count($string); $i++) {
    if ($string[$i] == "\n") { echo "\n"; continue; }

    $line = getLineNumber($text, $i);
    $col = getColumnNumber($text, $i);

    echo "Char: {$string[$i]} Ln: {$line} Col: {$col}\n";
}
```

Produces the following output:

Figure 10.14

```
Char: o Ln: 1 Col: 1
Char: n Ln: 1 Col: 2
Char: e Ln: 1 Col: 3

Char: t Ln: 2 Col: 1
Char: w Ln: 2 Col: 2
Char: o Ln: 2 Col: 3

Char: t Ln: 3 Col: 1
Char: h Ln: 3 Col: 2
Char: r Ln: 3 Col: 3
Char: e Ln: 3 Col: 4
Char: e Ln: 3 Col: 5
Char:   Ln: 3 Col: 6
Char: f Ln: 3 Col: 7
Char: o Ln: 3 Col: 8
```

```
Char: u Ln: 3 Col: 9
Char: r Ln: 3 Col: 10

Char: f Ln: 4 Col: 1
Char: i Ln: 4 Col: 2
Char: v Ln: 4 Col: 3
Char: e Ln: 4 Col: 4
```

For one-off uses, we can package our two functions into a set of string helper methods:

Figure 10.15 Line and Column Number Helper Method Implementations

```php
<?php

use Illuminate\Support\Str;
use Illuminate\Support\Stringable;

Str::macro('lineNumber', function ($value, $position) {
    return mb_substr_count(
            str($value)->substr(0, $position), "\n"
        ) + 1;
});

Stringable::macro('lineNumber', function ($position) {
    return Str::lineNumber($this->value, $position);
});

Str::macro('columnNumber', function ($value, $position) {
    $lastNlPosition = mb_strrpos(
        str($value)->substr(0, $position), "\n"
    );


    if ($lastNlPosition === false) {
        return $position + 1;
    }
```

```
25
26      return $position - $lastNlPosition;
27  });
28
29  Stringable::macro('columnNumber', function ($position) {
30      return Str::columnNumber($this->value, $position);
31  });
```

We have alluded that these helper methods might not be an optimal solution if we need to retrieve lines and columns frequently; this is mainly due to the large number of substrings we would be creating if we did so. To improve our existing implementations, we can create an index of all line endings within the text, similar to how we indexed character offsets when implementing our UTF-8 string iterator in Chapter 1.

Our implementation will use the single "\n" newline character. If the input text uses a different line ending style, we can use the normalizeEol helper method we have already developed. Using PHP's `preg_match_all` function with the `PREG_OFFSET_CAPTURE` flag:

Figure 10.16

```
1  <?php
2
3
4  $text = "one\ntwo\nthree four\nfive";
5
6  preg_match_all("/\n/u", $text, $matches, PREG_OFFSET_CAPTURE);
```

Would produce the following array of matches, with the positions of the newline characters within the text for us:

Figure 10.17

```
array:1 [
    0 => array:3 [
      0 => array:2 [
        0 => "\n"
        1 => 3
      ]
      1 => array:2 [
        0 => "\n"
        1 => 7
      ]
      2 => array:2 [
        0 => "\n"
        1 => 18
      ]
    ]
]
```

Our next task will be to convert the array of matches into a more straightforward array containing just the offsets of the newline characters. We can do this using Laravel Collections:

Figure 10.18

```php
<?php

// ...

$index = collect($matches[0])->map(function ($match) {
    return $match[1];
})->all();
```

Which produces the following array:

Figure 10.19

```
array:3 [
    0 => 3
    1 => 7
    2 => 18
]
```

Now, to get the line number from our index for any given position, we can iterate the index and compare the desired position to the positions within the index. Suppose the current position within the index is greater than the desired position. In that case, we know that the line number offset we are interested in was the previous offset within the index and can return the current value of the counter variable after adding one (to make our line numbers one-based):

Figure 10.20

```php
<?php

function getLineNumber($position, $index) {
    for ($i = 0; $i < count($index); $i++) {
        $offset = $index[$i];

        if ($i == 0 && $position < $offset) {
            return 1;
        }

        if ($offset > $position) {
            return $i + 1;
        }
    }

    return $i + 1;
}
```

We will do something similar with our new `getColumnNumber` implementation. Compared with the line number function, the difference with this implementation is

that we need to keep track of the value of the offsets. The loop's counting variable was enough to get the information we needed when counting line numbers. However, for columns, we need to know the position of the line ending within the string so we can compute our difference:

Figure 10.21

```
<?php

function getColumnNumber($position, $index) {
    $lastOffset = null;

    for ($i = 0; $i < count($index); $i++) {
        $offset = $index[$i];

        if ($i == 0 && $position < $offset) {
            return $position + 1;
        }

        if ($offset > $position) {
            return $position - $lastOffset;
        }

        $lastOffset = $offset;
    }

    return $position - $lastOffset;
}
```

Our two functions are pretty helpful, but it would become tedious if we manually initialize the index and pass it around whenever we wanted to use it. To get around this, we can create a new utility class to help out. Because naming things is incredibly difficult, we will call it `CharacterPositionLocator`:

Figure 10.22

```php
<?php

class CharacterPositionLocator
{
    protected $string = '';
    protected $newLineIndex = [];

    public function __construct(string $string)
    {
        $this->string = $string;
        $this->buildNewLineIndex();
    }

    protected function buildNewLineIndex()
    {
        preg_match_all(
          "/\n/u",
          $this->string,
          $matches,
          PREG_OFFSET_CAPTURE
        );

        $this->newLineIndex = collect($matches[0])
          ->map(function ($match) {
            return $match[1];
        })->all();
    }

    public function getLineNumber($position)
    {
        for ($i = 0; $i < count($this->newLineIndex); $i++) {
            $offset = $this->newLineIndex[$i];

            if ($i == 0 && $position < $offset) {
                return 1;
```

```
            }

            if ($offset > $position) {
                return $i + 1;
            }
        }

        return $i + 1;
    }

    public function getColumnNumber($position)
    {
        $lastOffset = null;

        for ($i = 0; $i < count($this->newLineIndex); $i++) {
            $offset = $this->newLineIndex[$i];

            if ($i == 0 && $position < $offset) {
                return $position + 1;
            }

            if ($offset > $position) {
                return $position - $lastOffset;
            }

            $lastOffset = $offset;
        }

        return $position - $lastOffset;
    }
}
```

We now have a utility class capable of retrieving line and column numbers from arbitrary positions, but we have another opportunity for refactoring. We will likely want the column number when we want to retrieve a line number, and we

can retrieve both pieces of information by slightly modifying the details of our `getColumnNumber` method. While doing these refactors, we can take the time to protect against supplying a negative position or one that exceeds the length of the string:

Figure 10.23

```
<?php

class Position
{
    public $position = null;
    public $column = null;
    public $line = null;
}

class CharacterPositionLocator
{
    protected $string = '';
    protected $length = 0;
    protected $newLineIndex = [];

    public function __construct(string $string)
    {
        $this->string = $string;
        $this->length = mb_strlen($string, 'UTF-8');
        $this->buildNewLineIndex();
    }

    protected function buildNewLineIndex()
    {
        preg_match_all(
          "/\n/u",
          $this->string,
          $matches,
          PREG_OFFSET_CAPTURE
        );
```

```
        $this->newLineIndex = collect($matches[0])
          ->map(function ($match) {
            return $match[1];
        })->all();
    }

    public function getLineNumber($position)
    {
        return $this->getCharPosition($position)?->line;
    }

    public function getColumnNumber($position)
    {
        return $this->getCharPosition($position)?->column;
    }

    public function getCharPosition($position)
    {
        if ($position > $this->length || $position < 0) {
            return null;
        }

        $lastOffset = null;
        $column = null;
        $line = null;

        for ($i = 0; $i < count($this->newLineIndex); $i++) {
            $offset = $this->newLineIndex[$i];

            if ($i == 0 && $position < $offset) {
                $line = 1;
                $column = $position + 1;
                break;
            }
```

```
            if ($offset > $position) {
                $line = $i + 1;
                $column = $position - $lastOffset;
                break;
            }

            $lastOffset = $offset;
        }

        if ($column === null) {
            $line = $i + 1;
            $column = $position - $lastOffset;
        }

        $charPosition = new Position();
        $charPosition->position = $position;
        $charPosition->column = $column;
        $charPosition->line = $line;

        return $charPosition;
    }
}
```

As part of our refactors, we defined a new Position class that will hold our desired line number and column number; it also stores our original position. To not break compatibility with our existing `getLineNumber` and `getColumnNumber` methods, and reduce the amount of duplicate code, we now internally make a call to our new method and return the appropriate member. We can reproduce the output of **Figure 10.14** using our new class like so:

Figure 10.24

```php
<?php

$text = "one\ntwo\nthree four\nfive";

$string = new Utf8StringIterator($text);
$locator = new CharacterPositionLocator($text);

for ($i = 0; $i < count($string); $i++) {
    if ($string[$i] == "\n") { echo "\n"; continue; }

    $pos = $locator->getCharPosition($i);

    echo "Char: {$string[$i]} Ln: {$pos->line} Col: {$pos->column}\n";
}
```

The next obvious question we may ask is can we do this reversed? Can we supply a line number and column and return the position within the string that corresponds to that location? So far, our implementation has assumed that the newline character present in the text will always be the "\n" sequence, but it could be any of the newline styles we are interested in supporting. Perhaps we may even see input text like the following, maybe as a result of poorly appended documents through some automated process:

Figure 10.25

```php
<?php

$text = "one\r\ntwo\nthree\r\nfour\nfive";
```

We can support this by changing our indexing regex to account for multiple newline styles. One side effect of many newline styles is that we will also need to store how many characters are part of the newline sequence. If we do not, we would not be able to easily account for position parameters that are inside a newline sequence. Additionally, suppose we will account for the possibility of many types of newline

sequences in a piece of text. In that case, we can also add a way to check if any given position belongs to a newline sequence. We will implement these changes before tackling retrieving positions from the known line and column numbers:

Figure 10.26

```
<?php

class Position
{
    public $position = null;
    public $column = null;
    public $line = null;
}

class CharacterPositionLocator
{
    protected $string = '';
    protected $length = 0;
    protected $newLineIndex = [];

    public function __construct(string $string)
    {
        $this->string = $string;
        $this->length = mb_strlen($string, 'UTF-8')
        $this->buildNewLineIndex();
    }

    protected function buildNewLineIndex()
    {
        preg_match_all(
~~            "/\n/u",~~
            "/\r\n|\n|\r/u"
            $this->string,
            $matches,
            PREG_OFFSET_CAPTURE
        );
```

```

        $this->newLineIndex = collect($matches[0])
          ->map(function ($match) {
              return $match[1];
              // Return an array containing the start
              // and end position of the sequence.
              return [$match[1], $match[1] + strlen($match[0]) - 1];
          })->all();
    }

    public function getLineNumber($position)
    {
        return $this->getCharPosition($position)?->line;
    }

    public function getColumnNumber($position)
    {
        return $this->getCharPosition($position)?->column;
    }

    public function isNewLine($position): bool
    {
        if ($position > $this->length || $position < 0) {
            return false;
        }

        for ($i = 0; $i < count($this->newLineIndex); $i++) {
            $nlStart = $this->newLineIndex[$i][0];
            $nlEnd = $this->newLineIndex[$i][1];

            if ($position >= $nlStart && $position <= $nlEnd) {
                return true;
            }
        }

        return false;
```

```
    }

    public function getCharPosition($position)
    {
        if ($position > $this->length || $position < 0) {
            return null;
        }

        $lastOffset = null;
        $column = null;
        $line = null;

        for ($i = 0; $i < count($this->newLineIndex); $i++) {
            $offset = $this->newLineIndex[$i];
            $offset = $this->newLineIndex[$i][1];

            if ($i == 0 && $position < $offset) {
                $line = 1;
                $column = $position + 1;
                break;
            }

            if ($offset > $position) {
                $line = $i + 1;
                $column = $position - $lastOffset;
                break;
            }

            $lastOffset = $offset;
        }

        if ($column === null) {
            $line = $i + 1;
            $column = $position - $lastOffset;
        }

```

```
104         $charPosition = new Position();
105         $charPosition->position = $position;
106         $charPosition->column = $column;
107         $charPosition->line = $line;
108 
109         return $charPosition;
110     }
111 }
```

We can utilize familiar techniques to implement our method to retrieve a character position for a line and column number. We will iterate each element of our newline character index and check whether or not the line matches the desired line. If it does, we will add the column number to the end position of the previous newline:

Figure 10.27

```
1  <?php
2  
3  class CharacterPositionLocator
4  {
5      // ...
6  
7      public function getCharPositionFromLineColumn($line, $column)
8      {
9          $position = null;
10         for ($i = 0; $i < count($this->newLineIndex); $i++) {
11             if ($i + 1 != $line) {
12                 continue;
13             }
14 
15             if ($i == 0) {
16                 return $column - 1;
17             }
18 
19             $index = $this->newLineIndex[$i - 1];
20 
21             $position = $index[1] + $column;
```

```
        }

        if ($position === null) {
            $lastIndex =
              $this->newLineIndex[count($this->newLineIndex) - 1];

            $position = $lastIndex[1] + $column;
        }

        return $position;
    }

    // ...
}
```

We can now use our new method to retrieve the position of the character that appears in line 5, column 3:

Figure 10.28

```
<?php

$text = "one\r\ntwo\nthree\r\nfour\nfive";

$locator = new CharacterPositionLocator($text);

// Returns 23
$locator->getCharPositionFromLineColumn(5, 3);
```

Now, this implementation works fine if the line and column numbers provided match a character within the boundaries of the line but can produce interesting results if we attempt to retrieve a character that does not. We will address this next.

The scenarios we will want to cover include

- handling negative line and column numbers,

- resolving positions that exceed the string's length, and
- column numbers that exceed the total number of characters for the requested line.

Figure 10.29

```
<?php

class CharacterPositionLocator
{
    // ...

    public function getCharPositionFromLineColumn($line, $column)
    {
        if ($line < 1 || $column < 1) {
            return null;
        }

        $position = null;
        for ($i = 0; $i < count($this->newLineIndex); $i++) {
            if ($i + 1 != $line) {
                continue;
            }

            if ($i == 0) {
                return $column - 1;
                $position = $column - 1;
                break;
            }

            $index = $this->newLineIndex[$i - 1];

            $position = $index[1] + $column;

            // Ensure the position is inside the current line.
            $nextLineStart = $this->newLineIndex[$i];
```

```
31
32              if ($position >= $nextLineStart[0]) {
33                  return null;
34              }
35
36              break;
37          }
38
39          if ($position === null) {
40              $lastIndex =
41                $this->newLineIndex[count($this->newLineIndex) - 1];
42
43              $position = $lastIndex[1] + $column;
44          }
45
46          if ($position >= $this->length || $position < 0) {
47              return null;
48          }
49
50          return $position;
51      }
52
53      // ...
54  }
```

We can update our sample code from **Figure 10.13** to include our refactored `getCharPositionFromLineColumn` method and print the results along with our previous output:

We should note that the following example assumes that the input string contains ASCII characters. To iterate each character of a multibyte string, we would need to change our loop to use our Utf8StringIterator class instance instead.

Figure 10.30

```php
<?php

$text = "one\r\ntwo\nthree\r\nfour\nfive";

$string = new Utf8StringIterator($text);
$locator = new CharacterPositionLocator($text);

for ($i = 0; $i < count($string); $i++) {
    if ($locator->isNewLine($i)) { continue; }

    $pos     = $locator->getCharPosition($i);
    $charPos = $locator->getCharPositionFromLineColumn(
        $pos->line,
        $pos->column
    );

    $printI       = str($i)->padLeft(2);
    $printCharPos = str($charPos)->padLeft(2);

    echo "Pos: {$printI} - {$printCharPos} ",
         "Char: {$string[$i]} Ln: {$pos->line} Col: {$pos->column}\n";
}
```

This results in:

Figure 10.31

```
Pos:  0 -  0 Char: o Ln: 1 Col: 1
Pos:  1 -  1 Char: n Ln: 1 Col: 2
Pos:  2 -  2 Char: e Ln: 1 Col: 3
Pos:  5 -  5 Char: t Ln: 2 Col: 1
Pos:  6 -  6 Char: w Ln: 2 Col: 2
Pos:  7 -  7 Char: o Ln: 2 Col: 3
Pos:  9 -  9 Char: t Ln: 3 Col: 1
Pos: 10 - 10 Char: h Ln: 3 Col: 2
Pos: 11 - 11 Char: r Ln: 3 Col: 3
Pos: 12 - 12 Char: e Ln: 3 Col: 4
Pos: 13 - 13 Char: e Ln: 3 Col: 5
Pos: 16 - 16 Char: f Ln: 4 Col: 1
Pos: 17 - 17 Char: o Ln: 4 Col: 2
Pos: 18 - 18 Char: u Ln: 4 Col: 3
Pos: 19 - 19 Char: r Ln: 4 Col: 4
Pos: 21 - 21 Char: f Ln: 5 Col: 1
Pos: 22 - 22 Char: i Ln: 5 Col: 2
Pos: 23 - 23 Char: v Ln: 5 Col: 3
Pos: 24 - 24 Char: e Ln: 5 Col: 4
```

2.2 Splitting Strings into Words

We can now split arbitrary strings into an array of lines, bringing us to our next logical step: breaking strings into lists of words. A word's concept and exact definition can easily lead us astray and into a separate discussion. With this in mind, we will say a space, or other characters we define (we will cover this later), separates a word. The first method we will create will be an adaptation of Laravel's `words` method. This approach will produce a list of strings that matches what the `words` helper method is operating on internally.

To do this, we can use the following regex, which will match any number of whitespace characters and split our string into an array of words:

Figure 10.32

```
/\s+/u
```

The regex in **Figure 10.32** will match any number of whitespace characters. We can now use this with `preg_split` to break our string into an array of words:

Figure 10.33

```
<?php

use Illuminate\Support\Str;
use Illuminate\Support\Stringable;

Str::macro('wordSplit', function ($value) {
    return preg_split("/\s+/u", $value);
});

Stringable::macro('wordSplit', function () {
    return collect(preg_split("/\s+/u", $this->value));
});
```

Calling our new `wordSplit` method on the `Str` class like so:

Figure 10.34

```
<?php

use Illuminate\Support\Str;

Str::wordSplit("A simple string made up of words.");
```

Would return the following result:

Figure 10.35

```
[
    "A",
    "simple",
    "string",
    "made",
    "up",
    "of",
    "words.",
]
```

Using our fluent macro would return a Collection instance with the same list of words. You may have noticed that the final element of the results in **Figure 10.35** contains a trailing full-stop (“.”).

We can address this by modifying our macro implementations to the following:

Figure 10.36

```
<?php

use Illuminate\Support\Str;
use Illuminate\Support\Stringable;

Str::macro('wordSplit', function ($value) {
    return preg_split("/\W+/u", $value, -1, PREG_SPLIT_NO_EMPTY);
});

Stringable::macro('wordSplit', function () {
    return collect(
        preg_split("/\W+/u", $this->value, -1, PREG_SPLIT_NO_EMPTY
      )
    );
});
```

The regex in **Figure 10.36** is slightly different, and it matches word boundaries, which include additional characters, not just whitespace. We are supplying -1 as the third

argument to instruct the `preg_split` function not to limit the number of elements it returns. We are passing the `PREG_SPLIT_NO_EMPTY` flag to have the function remove empty strings for us automatically.

Calling our modified methods on the following input:

Figure 10.37

```php
<?php

use Illuminate\Support\Str;

$string = "A simple string (word (word))
   made up of words; hello--there.";

Str::wordSplit($string);
```

Produces the following result:

Figure 10.38

```
[
    "A",
    "simple",
    "string",
    "word",
    "word",
    "made",
    "up",
    "of",
    "words",
    "hello",
    "there",
]
```

2.3 Counting Word Occurrences

While Laravel already provides a `wordCount` helper method, we could also use our `wordSplit` method in combination with PHP's `count` function:

Figure 10.39

```php
<?php

use Illuminate\Support\Str;

$string = 'The quick brown fox jumps over the lazy dog.';

// Returns 9
count(Str::wordSplit($string));

// Returns 9
Str::wordCount($string);
```

While using our custom method this way is not particularly advantageous, we can use it to accomplish some more nuanced tasks. One such thing might be to count the number of unique words in a string:

Figure 10.40

```php
<?php

use Illuminate\Support\Str;

// Returns 9
count(array_unique(
    Str::wordSplit($string)
));
```

The example in **Figure 10.40** doesn't seem to be working quite right, as it returns the same number of words as before. The culprit is the different casing between "The" and "the" within the sentence. We can solve this by transforming all of the words in our sentence first and then counting the result:

Figure 10.41

```php
<?php

use Illuminate\Support\Str;

// Returns 8
count(array_unique(
    array_map('mb_strtolower', Str::wordSplit($string))
));
```

Our code now returns the value 8, the number of distinct words in our sentence. The previous example works because of the call to PHP's `array_map` function. This function will apply a callback method to each element of an array; in our case, we will transform all of our strings to their lowercase variant. We can package these as new helper methods to be reused:

Figure 10.42

```php
<?php

use Illuminate\Support\Str;
use Illuminate\Support\Stringable;

Str::macro('uniqueWordCount', function ($value) {
    return count(array_unique(
        array_map('mb_strtolower', Str::wordSplit($value))
    ));
});

Stringable::macro('uniqueWordCount', function () {
    return count(array_unique(
        array_map('mb_strtolower', $this->wordSplit()->toArray())
    ));
});
```

What if we wanted to count the number of occurrences of each word? One way we could accomplish this is by using Laravel's Collection class. Let's think through

what it is we are doing. We are effectively grouping each lowercase word variant and counting the number of items that appear within that group. We can express this quite nicely using Collections:

Figure 10.43

```php
<?php

use Illuminate\Support\Str;

$string = 'The quick brown fox jumps over the lazy dog.';

collect(array_map('mb_strtolower', Str::wordSplit($string)))
    ->groupBy(fn($x) => $x)
    ->map(fn($g) => count($g))
    ->all();
```

Our code in **Figure 10.43** would produce the following result:

Figure 10.44

```
array:8 [
    "the"   => 2
    "quick" => 1
    "brown" => 1
    "fox"   => 1
    "jumps" => 1
    "over"  => 1
    "lazy"  => 1
    "dog"   => 1
]
```

At this point, we have created quite a few new helper methods to split our string into words, count the unique number of words, and count the number of times each word appears. However, we have two concepts of word splitting going on throughout these methods: retrieving *all* words and retrieving only the *unique* ones. Most of the code between these two techniques is shared, but we have no way to get a list of only the

unique words. We can address the latter issue and do some code cleanup by adding a `$unique` parameter to our existing `wordSplit` helper methods:

Figure 10.45

```php
<?php

use Illuminate\Support\Str;
use Illuminate\Support\Stringable;

Str::macro('wordSplit', function ($value, $unique = false) {
    if ($unique) {
        return array_unique(
                    array_map('mb_strtolower', Str::wordSplit($value))
                );
    }

    return preg_split("/\W+/u", $value, -1, PREG_SPLIT_NO_EMPTY);
});

Stringable::macro('wordSplit', function ($unique = false) {
    return collect(Str::wordSplit($this->value, $unique));
});

Str::macro('uniqueWordCount', function ($value) {
    return count(Str::wordSplit($value, true));
});

Stringable::macro('uniqueWordCount', function () {
    return Str::uniqueWordCount($this->value);
});
```

With these refactors out of the way, we can now define our new `wordFrequency` helper method to return the number of occurrences of each word in our text:

Figure 10.46

```php
<?php

use Illuminate\Support\Str;
use Illuminate\Support\Stringable;

Str::macro('wordFrequency', function ($value) {
    return collect(array_map('mb_strtolower', Str::wordSplit($value)))
        ->groupBy(fn($x) => $x)
        ->map(fn($g) => count($g))
        ->all();
});

Stringable::macro('wordFrequency', function () {
   return collect(Str::wordFrequency($this->value));
});
```

2.4 Finding Frequent and Repeated Words

In the previous section, we developed several word-related helper methods, among them being the wordFrequency method which returns the number of times a word appears in a piece of text. We can use this existing helper method to implement two new methods.

The first will find the top n most commonly used words in the text, and the other will find the words that repeatedly appear within the text. Both of these methods will be relatively straightforward to implement.

The first method we will implement will be the `commonWords` method, which will return an array of words up to some limit (the `$count` parameter). We do not care about the number of times the words appear in the text necessarily, just that the most commonly used ones appear first:

Figure 10.47

```php
<?php

use Illuminate\Support\Str;
use Illuminate\Support\Stringable;

Str::macro('commonWords', function ($value, $count = 10) {
    return collect(Str::wordFrequency($value))
        ->sortDesc()
        ->take($count)
        ->keys()
        ->all();
});

Stringable::macro('commonWords', function ($count = 10) {
    return collect(Str::commonWords($this->value, $count));
});
```

Using our method on the following input:

Figure 10.48

```php
<?php

use Illuminate\Support\Str;

$string = 'the the the of of hello world';

Str::commonWords($string, 2);
```

Would produce the following list of words:

Figure 10.49

```
array:3 [
    0 => "the"
    1 => "of"
    2 => "hello"
]
```

The `repeatedWords` method will be similar, but for this implementation, we want to ensure the words returned in our array appear at least once in the input:

Figure 10.50

```php
<?php

use Illuminate\Support\Str;
use Illuminate\Support\Stringable;

Str::macro('repeatedWords', function ($value, $count = 10) {
    return collect(Str::wordFrequency($value))
        ->filter(fn($x) => $x > 1)
        ->sortDesc()
        ->take($count)
        ->keys()
        ->all();
});

Stringable::macro('repeatedWords', function ($count = 10) {
    return collect(Str::repeatedWords($this->value, $count));
});
```

Calling `repeatedWords` like so:

Figure 10.51

```php
<?php

use Illuminate\Support\Str;

Str::repeatedWords($string, 3)
```

Returns:

Figure 10.52

```
array:2 [
    0 => "the"
    1 => "of"
]
```

2.5 Stop Words

So far, we have developed several helper methods for extracting words, counting their frequency, etc. A common issue arising when we perform text analysis or mining is the frequency of common words within natural languages. Consider the following sample text:

> The group of otherwise ordinary but eccentric teenagers ran towards the barn under destress, with nothing but a crumpled newspaper for covering, before the storm hit fully.

While we can get through this text relatively quickly, this English sentence contains many words or phrases that help "glue" the individual words together and help set the pacing. Using our `commonWords` helper method on this text produces the following list:

Figure 10.53

```
array:24 [
     0 => "the"
     1 => "but"
     2 => "group"
     3 => "of"
     4 => "otherwise"
     5 => "ordinary"
     6 => "eccentric"
     7 => "teenagers"
     8 => "ran"
     9 => "towards"
    10 => "barn"
    11 => "under"
    12 => "destress"
    13 => "with"
    14 => "nothing"
    15 => "a"
    16 => "crumpled"
    17 => "newspaper"
    18 => "for"
    19 => "covering"
    20 => "before"
    21 => "storm"
    22 => "hit"
    23 => "fully"
]
```

In the list of words produced, prepositions, coordinating conjunctions, and determiners dominate the top ten most commonly used words. We can help alleviate this problem through the use of stop words.

Stop words are a list of words we should remove from a piece of text before further processing. In our case, we will use a simple list of stop words. Still, the list of stop "words" can be anything that helps to make sense of the text we want to analyze: individual words, phrases, or combinations of them. There are many sources of stop

word lists, but for our purposes, we will use the following list of words to help clean up our text:

Figure 10.54

```
the
of
but
under
with
a
for
before
```

Our next step is to remove all of the words in our text that are also present in the stop word list. Suppose we wanted to describe our problem more formally. In that case, we could say that we have a set of words A and another B, and we want to produce a new set C containing all of the words in A that are not present in B.

We can accomplish this easily using Laravel's Collection features and the diff method. The diff method will allow us to filter a collection of items and return those that do not appear in whatever list we supply. The code in **Figure 10.55**:

Figure 10.55

```
<?php

use Illuminate\Support\Str;


$text = <<<EOT
The group of otherwise ordinary but eccentric teenagers
ran towards the barn under destress, with nothing but a
crumpled newspaper for covering, before the storm hit
fully.
EOT;

$words = str($text)->wordSplit(true)->diff([
```

```
    'the',
    'of',
    'but',
    'under',
    'with',
    'a',
    'for',
    'before'
]);
```

Would produce the following list of words:

Figure 10.56

```
group
otherwise
ordinary
eccentric
teenagers
ran
towards
barn
destress
nothing
crumpled
newspaper
covering
storm
hit
fully
```

Fantastic! We now have a method of removing words that add noise to our text analysis. Still, it would be even better if we could supply a list of stop words to our existing helper methods. Doing so would allow us to take advantage of things like word frequency, common occurrences, etc. while removing our stop words. **Figure 10.57** provides updated versions of our word-related helper methods with support for stop words:

Figure 10.57

```php
<?php

use Illuminate\Support\Str;
use Illuminate\Support\Stringable;

Str::macro('wordSplit', function ($value,
                                  $unique = false,
                                  $stopWords = []) {
    if ($unique) {
        return array_unique(array_map(
            'mb_strtolower',
            Str::wordSplit($value, false, $stopWords)
        ));
    }

    return collect(preg_split(
        "/\W+/u", $value, -1,
        PREG_SPLIT_NO_EMPTY
    ))->diff($stopWords)->all();
});

Stringable::macro('wordSplit', function ($unique = false,
                                         $stopWords = []) {
    return collect(Str::wordSplit($this->value, $unique, $stopWords));
});

Str::macro('uniqueWordCount', function ($value,
                                        $stopWords = []) {
    return count(Str::wordSplit($value, true, $stopWords));
});

Stringable::macro('uniqueWordCount', function () {
    return count(Str::wordSplit($this->value, true));
});

```

```
Str::macro('wordFrequency', function ($value,
                                      $stopWords = []) {
    return collect(array_map('mb_strtolower', Str::wordSplit($value)))
        ->diff($stopWords)
        ->groupBy(fn($x) => $x)
        ->map(fn($g) => count($g))
        ->all();
});

Stringable::macro('wordFrequency', function ($stopWords = []) {
   return collect(Str::wordFrequency($this->value));
});

Str::macro('commonWords', function ($value,
                                    $count = 10,
                                    $stopWords = []) {
    return collect(Str::wordFrequency($value))
        ->diff($stopWords)
        ->sortDesc()
        ->take($count)
        ->keys()
        ->all();
});

Stringable::macro('commonWords', function ($count = 10) {
    return collect(Str::commonWords($this->value, $count));
});

Str::macro('repeatedWords', function ($value,
                                      $count = 10,
                                      $stopWords = []) {
    return collect(Str::wordFrequency($value))
        ->diff($stopWords)
        ->filter(fn($x) => $x > 1)
        ->sortDesc()
        ->take($count)
```

```
        ->keys()
        ->all();
});

Stringable::macro('repeatedWords', function ($count = 10) {
    return collect(Str::repeatedWords($this->value, $count));
});
```

We can now use our existing wordFrequency helper method to retrieve all words that were are interested in, with their number of occurrences:

Figure 10.58

```
<?php

use Illuminate\Support\Str;

Str::wordFrequency($text, [
    'the',
    'of',
    'but',
    'under',
    'with',
    'a',
    'for',
    'before'
]);
```

2.6 Constructing Initialisms and Acronyms

An interesting thing we can do with the word helper methods is to create a new one to help construct initialisms or acronyms from a string of words. For example, we may want to shorten the name “Jane Doe” to “JD” or even “J.D.” We already have a way to retrieve a list of words from a string using `wordSplit`. Our basic process will

be to iterate each of the words of our string and extract the first character from each word and then concatenate them.

Leveraging our existing helper method and Laravel's Collections, this simple implementation could look like the example in **Figure 10.59**:

Figure 10.59

```php
<?php

use Illuminate\Support\Str;

// Returns "JD"
collect(Str::wordSplit('Jane Doe'))
    ->map(fn($w) => mb_substr($w, 0, 1))
    ->join('');
```

The implementation in **Figure 10.59** works for our simple text input, but what if we wanted only to include words that have their first letter upper-cased? To do this, we can grab the first character of each word and check to see if it is already capitalized and if so, we will use it. We will start implementing these additional features in an `initials` macro method:

Figure 10.60

```php
<?php

use Illuminate\Support\Str;

Str::macro('isCapitalized', function ($value) {
   if (str($value)->isEmpty()) {
         return false;
   }

    $firstChar = str($value)->substr(0, 1);

   return $firstChar == mb_strtoupper($firstChar);
});
```

```
14
15 Str::macro('initials', function ($value, $ucOnly = true) {
16     return str($value)->wordSplit->reject(function ($word)
17                                                 use ($ucOnly) {
18         if ($ucOnly && !Str::isCapitalized($word)) {
19             return true;
20         }
21
22         return false;
23     })->map(function ($word) {
24         return mb_strtoupper(mb_substr($word, 0, 1));
25     })->join('');
26 });
```

We have a lot going on with this updated implementation. To start, we added a new helper method `isCapitalized` that tests if the provided value has its first character uppercased. We use this inside a newly added "reject" callback on our collection of words.

When the reject callback returns true, it will not include that word in the final list. We are taking advantage of this to exclude any words that do not have their first letter capitalized. Finally, we modified the final map callback to uppercase all the characters in the final initialism.

Calling our new helper method from **Figure 10.60** would produce the following results:

Figure 10.61

```
1 <?php
2
3 use Illuminate\Support\Str;
4
5 // Returns "JD"
6 Str::initials('Jane m Doe');
7
8 // Returns "JMD"
9 Str::initials('Jane m Doe', false);
```

2.7 Calculating Estimated Reading Time

An interesting thing we can do with our word helper methods is to calculate an estimated reading time for a piece of text. From an implementation standpoint, it is relatively straightforward: we need to know the number of words within the text and have a value for the average number of words people read in some time frame.

An analysis of 190 studies estimates that the average reading time was between 238 and 260 words per minute, depending on the genre[1]. This estimate was arrived at by looking at the reading time of adults silently reading English text. For our use case, we will use the lower value of 238 words per minute for non-fiction in our calculations.

Our first pass at implementing this feature may look something like **Figure 10.62**:

Figure 10.62

```
1  <?php
2
3  use Illuminate\Support\Str;
4
5  Str::macro('readingTime', function ($value, $wpm = 238) {
6      return ceil(Str::wordCount($value) / $wpm);
7  });
```

Our implementation uses the wordCount helper method to retrieve the number of words in the provided value and divides it by the desired words per minute. We also use PHP's ceil function on the result to round up the nearest whole minute. These calculations are estimates at best, and it is not entirely practical to break things down into minutes and seconds in most cases.

We can use our new `readingTime` helper method like so to get the approximate reading time of a string containing 1,000 words:

[1]Marc Brysbaert, "How Many Words Do We Read per Minute? A Review and Meta-Analysis of Reading Rate," Journal of Memory and Language 109 (2019): p. 104047, https://doi.org/10.1016/j.jml.2019.104047.

Figure 10.63

```php
<?php

use Illuminate\Support\Str;

// Returns 5.0
Str::readingTime(
    Str::repeat('word ', 1000)
);
```

Our method returns an approximate value of five minutes. The next question might be, what about if we had an extremely long piece of text? For example, using our default words per minute value, a text containing 65,535 words would return an estimated reading time of 275 minutes. Most people are comfortable with smaller units of time and may find it challenging to understand precisely how long 275 minutes is quickly. A better user experience might be to break these larger time frames into smaller units, such as hours and minutes.

We could work to implement this ourselves, but one of Laravel's dependencies is the Carbon date/time library. Luckily for us, a `CarbonInterval` class exists that allows us to take a time interval expressed in seconds and convert it into a more understandable format.

The following refactors to our implementation to return a new `CarbonInterval` instance, which provides many helpful methods (note that we are multiplying our previous results by sixty to convert the number of minutes into seconds):

Figure 10.64

```php
<?php

use Illuminate\Support\Str;
use Illuminate\Support\Stringable;
use Carbon\CarbonInterval;

Str::macro('readingTime', function ($value, $wpm = 238) {
    return CarbonInterval::seconds(
        ceil(Str::wordCount($value) / $wpm) * 60
    )->cascade();
});

Stringable::macro('readingTime', function ($wpm = 238) {
    return Str::readingTime($this->value, $wpm);
});
```

The `cascade` method call on the `CarbonInterval` class is what will break our seconds into smaller units. Using our updated helper method in **Figure 10.64**, we can now chain the `forHumans` method call to output nicely formatted results:

Figure 10.65

```php
<?php

// Returns "4 hours 36 minutes"
str('word ')
    ->repeat(65535)
    ->readingTime->forHumans();
```

We've covered a lot of ground in this chapter, and there is still much more to do with lines and words. You may ask: "Where do we go from here?". We go to chapter eleven.

Thank you for taking the time to read this sample! If you are interested in reading more, please consider purchasing a copy of the book!