MASTERING RUBY

STRINGS AND ENCODINGS

AARON LASSEIGNE

Mastering Ruby
Strings and Encodings

Author

Aaron Lasseigne

Editor

Taylor Fausak

Edited by Taylor Fausak

Version: 1.0.4

Published: 2017

This book is dedicated to my wife and daughter who encouraged me and gave up their time with me so that I could pursue this. I love you both.

I'd also like to thank everyone who helped along the way. It reminds me that my life is rich with friends who support me.

Contents

Forward

The internet is powered by multimillion dollar string manipulation machines. We put strings in a box, and get new strings out. While there's plenty of mathy things that can happen in the middle, there is no denying the importance of strings in today's world.

You might think you know how to work with strings, but do you know how strings work? Maybe you've never thought much about it, taking for granted the tremendous power of this simple construct as provided by Ruby. Maybe you've read a bit about internal data structures, but overlooked the string. Maybe this is your first attempt at digging deeper into one of the core foundations holding up the modern web. No matter the reason you've picked up this book, you are in the right place.

I'm Richard Schneeman. I'm in the top 50 contributors to Ruby on Rails and I work for Heroku. My focus is on usability and performance and when it comes to performance in Ruby there is one golden rule: fewer objects means faster applications. When it comes to Ruby applications, many objects are strings. That's why I was excited to get an advanced copy of Aaron's book "Mastering Ruby: Strings and Encodings" and completely floored when I found brand new information within the first few pages.

I've been using Ruby for over ten years, and I'm still surprised to find new corners to explore. It wasn't until I came across this book that I realized how deep the string "rabbit hole" went. While building and manipulating strings is one of the very first things programmers are taught to do (think 'puts "hello world"') we quickly "graduate" to more advanced structures such as Arrays and Hashes. This book is a homecoming of sorts, a call to the roots of our own programming origins. The surface of this simple data structure might look calm and serene, but it holds a tremendous depth that I can just about guarantee you've not fully explored.

While "Mastering Ruby: Strings and Encodings" has plenty of practical high level Ruby examples (including how to make a quick and effective text only progress bar), I found the depth and detail to be extremely helpful. I find the more I know about my tools, the more effective I am at using them, and strings and encodings are no different.

If you're like me, then you've not had the benefit of any formal computer science undergraduate education. If that's the case then Aaron has your back. He expertly explains bit by bit the science and theory behind how strings are represented on disk. This book starts off at the dawn of the computer age and through a blend of history, example, and great writing shows how strings are represented internally. Even if you've got a CS background, the anecdotes and origins may be quite new to you, and the fast paced but easy to read style will serve as a good refresher.

All of the theory and the science in the world won't help unless it can be practically applied. That's why after getting the basics of storage and encoding out of the way, Aaron goes straight into examples of string use and manipulation. Some of the tips are curious oddities left over from bygone days of Ruby 1.8.7 past, but many more are practical and informative. Just when I thought I knew all there was to know about a specific method or topic like Heredocs, Aaron found yet another way to surprise me.

If you've been using Ruby for awhile you'll be delighted at all the hidden corners exposed in this book. If you're just getting out, the examples and documentation are quite exhaustive. Either way, after you read this book, you'll be better and faster at working with strings in Ruby.

Richard Schneeman

1 | Strings?

I wish I could say it came to me in a dream. In reality, I was sitting at my desk, drumming my fingers, and thinking. Something had occurred to me: ==strings are the most important and prolific data type.==

I've stored more information in strings than anything else. Sometimes I'd used them correctly for storing labels like a person's name or a book title. Other times I'd used them lazily in place of a more fitting object. Why build out an object to handle the intricacies of email addresses when I could just use a string? I didn't need access to specific parts of the email and if I did I could use a regular expression to get it. My problem was adequately solved and time was saved.

Strings often act in this capacity as a sort of omni-type. I'd even heard people pejoratively call it "string-typing". When I did go the distance and create value objects they were almost always strings under the hood. Their flexibility and usefulness is undeniable. This is why we find strings are everywhere. Console output, console input, web parameters, serialization, binary data... strings, strings, and more strings. I'd venture to guess that programmers have created more strings than any other object.

How well do we know them?

It's hard to pin down exactly but the term "string" seems to be a shortening of "string

of characters". It was used in the same way a person might use a "string of pearls" or a "string of beads". By the early '60s people were just calling them "strings". It turns out programmers have been lazy for a while now.

Early strings were simple. It didn't take much to understand them. Today there's so much more to know. We have encodings, normalization forms, interpolation, etc. Do you know what a code point is? How about the difference between `%q` and `%Q` or why `'à'.size` returns `2`?

Strings are letters, numbers, whitespaces, punctuations, symbols, and emoji smashed together into a symphony of information. We want to understand how they work and use them in the best way possible. To do that, we're going to start from the bottom and work our way up. As we progress through the chapters we'll layer on more and build off what we know. By the end you'll know more about strings than you thought possible.

2 | First Byte

How do you code a whale? One byte at a time. - programming proverb

Strings appear as characters but they lie. They're one of the few constructs that present to the programmer almost exactly as they present to the audience. Most of the time it's the right thing to do. No one wants to try to read a string by its internal representation. Just show me what it says! At times, this convenience disguises the truth in problematic ways. Is that character a semi-colon (";") or a Greek question mark (";")?

When we run into these issues, and believe me we will, understanding how bytes turn into characters will be critical. Knowing how to examine bytes will make for a fantastic diagnostic tool. It'll also help if you ever find yourself working with binary data which, in Ruby, is represented as a string.

Understanding bytes starts with an understanding of how computers think. To a computer, the world is made of electrons and switches. Data is stored, processed, and transmitted as a stream of ones and zeros. Ons and offs.

on, off, on, off, on, off

101010

These ones and zeros are all the computer needs. Everything else is a convenience

3

for us. Layers of abstraction built to make our lives easier. We're going to peel back the abstraction and in doing so, begin the journey toward mastery.

Bits and Bytes

We'll start with the binary digit. The bit. The smallest unit of computing. A physical switch that's either off (0) or on (1). These switches have always been there but they've shrunk over the decades. Today we even have designs that hold multiple bits on a single switch (multi-level cell floating-gate transistors). To give you an idea of how tiny they've become, a 512GB hard drive has over four trillion bits of storage. We'll start our discussion with eight bits.

One bit at a time isn't a practical way to work. A more useful approach is to group and treat them as a single unit of information. Early on, computers varied in their chosen group size. Some grouped six or seven bits but the most popular size was eight. These bundles of bits are called bytes. It doesn't matter how many bits there are, it's still a byte. Today, we assume all bytes are composed of eight bits because, well, they basically are. It would be more accurate to call it an "octet" but accuracy has never stopped the evolution of language.

These bytes are bundles of information but we pretend they're numbers. It's easier to describe a byte or series of bytes with numbers rather than a reading of each bit along the way. When we see 00000000 we say "0", 00000001 is "1", 00000010 is "2", and so on. Each digit in the sequence is a bit so it can only be 0 or 1. That's why we represent "2" as 00000010. To translate between the number and the bits described we'll need to know how binary numbers work.

With a few exceptions, everyone on the planet learns to use decimal numbers. That means a single digit can be any number between 0 and 9. It's also called "base ten" because each digit can represent one of ten possible values. Advancing past 9, the

highest value, requires a second digit. When we use two digits the one to the left is worth ten times the amount shown. We can read the number 23 as 2 tens and 3 ones. Each time we add another digit we multiply its worth by another ten times. We could read 472 as 4 hundreds, 7 tens, and 2 ones.

Binary numbers follow this exact same system but use a base of two. Each digit can represent any number as long as it's 0 or 1. After 1 we'll have reached the highest value for a single digit and we'll need a second digit. With base ten, the second digit is worth ten times the value shown. For base two it's two times the value shown. In binary, when we write 10, it's the same as saying 1 two and 0 ones. Each new digit is worth two times more. To write five we'd use 101 which is 1 four, 0 twos, and 1 one. Our decimal 23 from before is 10111 in binary. We get to 23 with 1 sixteen, 0 eights, 1 four, 1 two, and 1 one.

This same principle applies to octal numbers (base eight) and hexadecimal numbers (base sixteen). With hexadecimal, when we pass 9 we switch over to using letters. Ten through sixteen are mapped to "a" through "f".

For certain situations it's easier to use one of these other bases. Ruby is nice enough to provide a mechanism to do just that. Any number starting with a 0 followed by b (binary), o (octal), or x (hexadecimal) will be correctly interpreted.

```
>> 0b10111 # binary
=> 23

>> 0o27 # octal
=> 23

>> 0x17 # hexadecimal
=> 23
```

Ruby keeps us happy by focusing on decimals. Did you notice that every output above is the decimal equivalent of the input? We don't have the mental framework for these other bases and Ruby knows it. Don't get me wrong, I'm sure with some

practice you could learn to consistently read other bases. The trick will come when you try to discuss them.

Languages the world over have systems dedicated to decimals. We write them in groups with separators. We have words to explain which power of ten a digit represents. Say the following two sentences aloud (especially if you're on a bus full of people and it would be super awkward):

"We sold five zero zero one four three one units today!"

"We sold five million, one thousand, four hundred, thirty-one units today!"

If you walked up to me and said the first one, I'd assume you were a robot from the future sent to kill me and I'd run. We see the same issue when we read or write numbers.

5001431 vs 5,001,431

Which one is easier to read? Ask some meat beings and they'll all pick the second one. Ruby won't add these separators but it will give you decimals. Remember that these decimals are labels for the underlying bit structure. They label a piece of data which could represent anything. Even a decimal! In the future we'll see 65 (i.e. 01000001) used to represent the letter "A". It's all about the data.

Taking a Byte

We've got a string. We know it's made with bytes. How do we get those bytes? We follow the rule of least surprise and call `bytes`.

```
>> 'abc'.bytes
=> [97, 98, 99]
```

You'll notice that we have three characters and three bytes. It's no coincidence. Each of these characters fits into one byte. That won't always be the case. To start off we'll keep it simple and only use single byte characters.

We can loop over the bytes with either `bytes.each` or `each_byte`. Using `each_byte` will be slightly faster. As a general rule in Ruby, if a method name is a combination of two other method names, it's probably an optimization. We can see this with other methods like `each_key` and `reverse_each`.

```
>> 'abc'.each_byte do |byte|
..   puts byte
.. end
97
98
99
=> "abc"
```

To get a single byte we'll call `getbyte` with the index of the byte we want.

```
>> 'abc'.getbyte(0)
=> 97
>> 'abc'.getbyte(1)
=> 98
>> 'abc'.getbyte(2)
=> 99
```

We can even grab from the back with negative indexes.

```
>> 'abc'.getbyte(-1)
=> 99
>> 'abc'.getbyte(-2)
=> 98
>> 'abc'.getbyte(-3)
=> 97
```

If we go too far in either direction we'll get back `nil`.

```
>> 'abc'.getbyte(3)
=> nil
>> 'abc'.getbyte(-4)
=> nil
```

We can also find the number of bytes by calling `bytesize`. We're using only single byte characters right now so it's no different than calling `size`. With multi-byte characters, the difference is clear.

```
>> 'abc'.bytesize
=> 3
>> 'àbç'.bytesize
=> 5
```

Setting Bytes in a String

Twiddling individual bytes is easy with `setbyte`. Like `getbyte` it'll take a positive or negative index followed by a replacement byte.

```
>> word = 'hello'
=> "hello"
>> word.setbyte(0, 'c'.ord)
=> 99
>> word
=> "cello"
```

When using `setbyte` we must exercise caution. To start with, it mutates the string it's called on. It also behaves peculiarly when called with integers that are too large.

With `setbyte` we're altering a single byte. The largest value for a single byte is 11111111 or in decimal terms, 255. Any higher number would require multiple bytes. What happens if we give it 355?

```
>> word = 'hello'
```

```
=> "hello"
>> word.setbyte(0, 355)
=> 355
>> word
=> "cello"
```

I bet you didn't see that coming. It turns out `setbyte` performs a modulus against the incoming value to ensure a fit. Our 355 undergoes a modulus 256, turns into 99, and we get a "c".

Frankly, there are safer ways to swap out a character. Imagine the damage we might cause swapping a single byte in a multi-byte character.

```
>> e = "é"
=> "é"
>> e.setbyte(0, 65)
=> 65
>> e
=> "A\xA9"
```

The results will be... interesting.

Byteslice

From the name you might think `byteslice` would let us extract a series of bytes. No one would blame you for being wrong. A vestige of a bygone era when single byte characters reigned supreme, this method is one to be avoided. In truth, it takes a series of bytes and returns the characters they represent.

```
>> 'résumé'.byteslice(0)
=> "r"
```

There are two ways to select characters for extraction. We can use a position and

length or a `Range`. With a `Range`, the bounds are the starting and ending positions. A moment ago we extracted one byte and got the "r" in "résumé". The "é" in "résumé" is two bytes. If we want the first two letters we'll need to grab three bytes.

```
>> 'résumé'.byteslice(0, 3)
=> "ré"
>> 'résumé'.byteslice(0..2)
=> "ré"
```

What if we only grab two bytes?

```
>> 'résumé'.byteslice(0, 2)
=> "r\xC3"
```

Ruby can't print half an "é" so it gave us "\xC3". What does that mean? Let's look at what bytes make up "é".

```
>> 'é'.bytes
=> [195, 169]
```

We only grabbed the first byte so all we got was `195`. Ruby couldn't find a character represented by `195` so it printed "\xC3". The "\x" indicates a hexadecimal value. The "C3" that follows is the hexadecimal representation of the byte. We can check this by typing `0xC3` into an IRB session or by converting the string "C3" to an integer. With `to_i` we can pass the base we want to convert from. In our case we're converting from hexadecimal which is base sixteen.

```
>> 'C3'.to_i(16)
=> 195
```

I mention `byteslice` so that you know what it does. Seeing the name alone might lead you to believe this method is useful. It is not. Its behavior is odd and nonintuitive. If you run across it I recommend a refactor with extreme prejudice.

Recap

Let's go over what we know. Bits are the smallest unit of computing. By grouping bits we create bytes. These bytes are used to store everything we work with.

With `bytes` we can see what bytes are in a string. We can read and set individual bytes with `getbyte` and `setbyte`.

We know that bytes can represent characters but how do we get from here to there?

3 | Encodings

Why does 65 (i.e. 01000001) mean "A"? Do all computers know that 65 means "A"? Strings are made from bytes but we need to know how those bytes are translated into characters. To do this we need a second piece of data. We need an encoding.

Encodings are maps that translate nonsense into meaning. They're shared standards that explain how to break down bytes and make them into characters. Every Ruby string has an encoding.

```
>> 'abc'.encoding
=> #<Encoding:UTF-8>
```

We'll start off by looking at the most influential encoding ever. Even with limited functionality it makes appearances decades after its creation. It's at the base of many modern encodings including the most popular in use today. If there was a museum of encodings, this one would have the largest exhibit.

ASCII

The American Standard Code for Information Interchange (ASCII) was first published in 1963. At the time it defined 99 character codes. Some of those codes were actual characters like "A". Others were control codes used to adjust format-

ting and facilitate communication between machines. Most of these control codes have become obsolete. The original standard didn't even include lower-case letters. EVERYONE JUST YELLED.

ASCII was designed to be a fixed-width single-byte encoding. It was created to replace existing encodings on teleprinters. A teleprinter was a keyboard, printer, and phone packaged in one box. Two teleprinters would be connected over a phone line and then they could send messages back and forth. Typing on one caused the other to print the message. One of the first to use ASCII was the Teletype Model 33.

Photo By: Rama & Musée Bolo

ASCII had a limited character set and only needed 7 bits to represent all of it. There was no reason to add more bits to the data. Anything larger would result in wasted storage and extra bits being transmitted down the wire. These machines could be made to use 7-bit bytes so ASCII was created as a 7-bit encoding. An update to ASCII was published in 1967 that added lower-case characters and made a few minor changes to some symbols. Even with these additions they still didn't need more than 7 bits.

The teletype machines were hooked into widely popular "minicomputers", a term that today seems ridiculous. These "minicomputers" were 100 pound behemoths and at the time would have cost \$15,000 to \$20,000. Grab your pocket computer that also makes phone calls and ponder that for a minute.

Photo By: Stefan Kögl

In 1968, the National Bureau of Standards (today it's called the National Institute of Standards and Technology) adopted ASCII as their first Federal Information Processing Standard. President Lyndon B. Johnson signed off on the new standard. Going forward government computers and related equipment would need to support ASCII.

Widespread use along with support from the U.S. government and from major companies like IBM helped to cement ASCII as the encoding of the future. ASCII proved so popular that its reign lasted well into the internet era. Even without international characters, it was the most popular webpage encoding until 2008.

During this time computers shifted toward 8-bit bytes. Adapting ASCII to this bit world was simple enough. A leading zero was added and everyone carried on.

The lack of international support was becoming an issue. ASCII was only designed with English in mind. It turns out much of the world uses characters not contained within the English alphabet. Not only that, they wanted to use these other characters on computers. Then one day someone realized that ASCII only used 7 of the 8 bits in a byte. Half of the possible values were going unused. That empty half could be filled with the characters needed for any particular locality. Welcome to ASCII fragmentation.

"Extended" ASCII variants began popping up everywhere. Accented vowels, currency markers, and additional punctuation were all common additions. Most maintained backwards compatibility by only adding new characters. Some went so far as to reach back into the ASCII set and make small adjustments.

One popular extension was the work of the International Organization for Standardization (ISO). (No astute reader, I did not mess up that acronym. That's what happens when multilingual countries get together, use local names to represent the same group, and compromise by picking an acronym that doesn't work for any of them.)

In 1987, ISO 8859-1 was published. It added characters to handle western European languages and removed the control codes. Five years later, in 1992, a modified version was released that added the control codes back in. The new version was called ISO-8859-1. Note the dash following "ISO" which allows you to easily distinguish the two with almost no chance of confusing them. This character set occasionally goes by the name Latin-1.

Over the years, ISO released many more ASCII extensions. Each was valuable to a particular group but remained incompatible with the other extensions. Other groups created even more extensions and eventually ASCII was renamed to US-ASCII to clarify things a bit. For a while this hodgepodge worked. It eventually became clear that these encodings weren't going to fix the larger issue. We needed a single encoding to handle all the languages of the world. It would need to store tens of thousands of characters. Ideally it would avoid breaking all of the existing ASCII support.

How would we go about building such a panacea?

4 | Greater than 256

Let's take a moment to review. Not everyone in the world speaks English. Even though they don't speak English, through hand signals and hard work we've determined that they want in on this encoding craze. Our new encoding will need to handle more than the 256 characters we get using one byte.

Let's see if we can solve this.

Attempt #1

It's clear that we'll need more than one byte. Let's add a second byte and see where that takes us.

A second byte ups our storage from 256 possible characters to a whopping 65,536. Surprisingly, it won't be enough. There are a lot of characters in the world. We'll stick with two bytes for now but if this approach pans out we'll need to add more.

Our new encoding is a 2-byte fixed-width encoding. Each character is going to use exactly two bytes. Let's call our encoding ASCII-2B for two bytes.

To start we'll drop the existing ASCII characters in. Each one will be prefixed with a byte of all zeros. We'll add the zero-filled byte in front so that existing characters

retain the same value. Here's what an "A" looks like in ASCII:

128	64	32	16	8	4	2	1
0	1	0	0	0	0	0	1

In ASCII-2B we'll prepend a byte of all zeros:

32768	16384	8192	4096	2048	1024	512	256
0	0	0	0	0	0	0	0

128	64	32	16	8	4	2	1
0	1	0	0	0	0	0	1

In our mythical encoding, calling `bytes` would return two values.

```
>> 'A'.bytes
=> [0, 65]
```

Now that ASCII is taken care of we need to expand. Everything above 127 is wide open.

ASCII-2B will require us to re-implement a lot of the methods on `String`. The changes turn out to be fairly easy. We only have to account for two bytes instead of one. In ASCII the `size` function can be an alias of `bytesize` since each character is one byte. Our version will divide by the number of bytes per character.

```
BYTES_PER_CHARACTER = 2

def size
  bytesize / BYTES_PER_CHARACTER
end
```

I mentioned earlier that we're trying out two bytes per character but we'll need more. With `BYTES_PER_CHARACTER` our solution can be easily adjusted to three or more bytes as needed. As long as the characters are all the same width, we're good.

Accessing individual characters using `[]` isn't bad either. (There's a lot more `[]` can do and we'll cover it in future chapters. For now, we'll ignore everything else and only implement character retrieval.)

```ruby
def [](char_index)
  bytes
    .slice(char_index * BYTES_PER_CHARACTER,
      BYTES_PER_CHARACTER)
    .each_with_object(' ')
    .with_index do |(byte, placeholder), i|
      placeholder.setbyte(i, byte)
    end
end
```

We'll use `slice` to take one character worth of bytes starting at `char_index * BYTES_PER_CHARACTER`. We're starting twice as far into the byte array because each character takes two bytes. Once we have the bytes we need, we'll use a placeholder character and change it for the new character with `setbyte`. Here we're using a space character as the placeholder. Every character has the same number of bytes so it doesn't really matter which one we use. We replace each byte in the placeholder and at the end we'll have our new character string that matches the requested character.

You can imagine how other methods would undergo similar changes. Fixed-width encodings are easy to understand and relatively simple to implement. Unfortunately, it's not all good news.

With ASCII-2B we've doubled the space needed to store and transmit our text. If the majority of your characters are ASCII then we've doubled it and gained nothing. You might argue that storage has gotten cheap. Turning a 10K file into a 20K doesn't

seem too bad. The problem is when you start to think about how much text we use.

Are you ready for all of your HTML, CSS, JavaScript, and JSON to double in size? Imagine the additional strain on networks. The vast majority of characters in those files are from the ASCII set. They're being doubled in size when they don't need to be.

There's also the problem of all the existing ASCII code. Converting text from ASCII to ASCII-2B is a trivial operation. The code to work with ASCII-2B is similarly easy. The problem is trying to coordinate a change like that.

We can't possibly get everyone to change at once. It would have to be an incremental change as systems updated slowly over time. Computers could work with both encodings but many standards aren't designed for that. For example, text files (.txt) lack an encoding and are assumed to be ASCII.

Maybe we can come up with a solution that's less disruptive.

Attempt #2

How can we continue to support single byte ASCII characters but have multiple bytes for newer characters? We'll have to take a new approach. Using fixed-width characters is out. Instead we're going to create an encoding that has some 1-byte characters and some multi-byte characters. We're going to do it by leveraging one little bit.

In the Encodings chapter, we talked about how ASCII is really a 7-bit encoding. In our 8 bit systems the leading bit is always zero. This is the bit everyone jumped on to create all of those ASCII extensions. Unlike them, we won't use it to store more values in the first byte. We'll turn it into a flag.

All ASCII characters will go in as they are with the leading `0` bit. New characters will be two bytes long and the first byte will have the leading bit set to `1`. We can repeat this pattern for as many bytes as we need. A leading bit of `1` will tell us to include the next byte. If the leading bit is set to `0` then we stop.
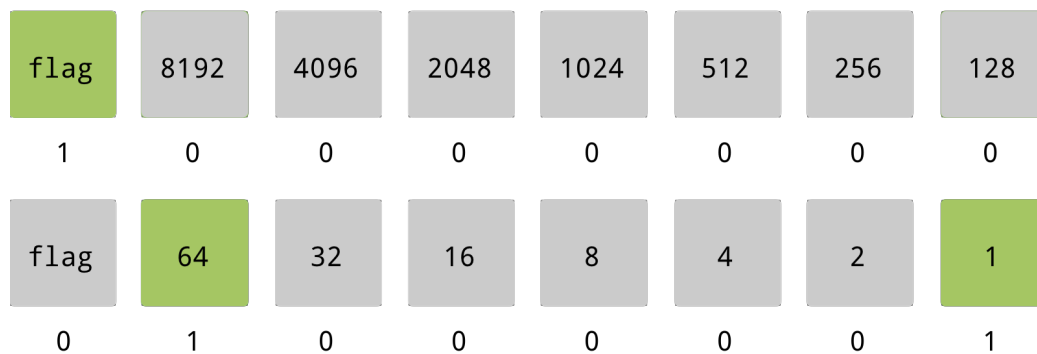
Our "A" is the exact same as the ASCII version.

flag	64	32	16	8	4	2	1
0	1	0	0	0	0	0	1
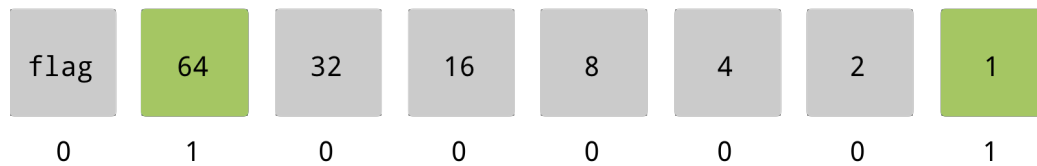
We can add "Ä" by using two bytes.

flag	8192	4096	2048	1024	512	256	128
1	0	0	0	0	0	0	0

flag	64	32	16	8	4	2	1
0	1	0	0	0	0	0	1

With this approach each byte holds less data. A single byte uses 7 bits to store 128 characters. Two bytes uses 14 bits to store another 16,384 characters. If we go to three bytes we get 21 bits and can store an additional 2,097,152 characters. Our variable-width encoding can use between one and three bytes to store up to 2,113,664 characters. We get less data per byte but it grows massively as we add bytes and we can add as many as we like. Because of its variable byte approach we'll name this one ASCII-VB.

We've built a backwards compatible encoding that can store every character we'll ever need. Victory! There are some negatives that we need to address.

With ASCII-2B it was easy to calculate the size of the string. In ASCII-VB it's going to be more difficult. We can't calculate it from the number of bytes like before. We'll need to walk the bytes and count how many characters there are. There's one piece of information that will help us. We know that every character will have exactly one byte with a leading 0. We'll call it the "terminal byte". Terminal bytes are easy to find because they have a value less than 128. That's the most you can fit in 7 bits. We can figure out how many characters there are by counting the number of terminal bytes.

```ruby
def size
  bytes.select { |byte| byte < 128 }.size
end
```

It's trickier than the ASCII-2B version and more CPU intensive but it's not too bad. Now let's try the same accessor method we made for ASCII-2B.

```ruby
def [](char_index)
  bytes_grouped_by_char = bytes
    .slice_after { |byte| byte < 128 }
    .to_a

  char_bytes = bytes_grouped_by_char[char_index]

  char_bytes
    .each_with_object(' ' * char_bytes.size)
    .with_index do |(byte, placeholder), i|
      placeholder.setbyte(i, byte)
    end
end
```

The code is clearly more complicated than before. We start by taking the bytes and grouping them up with `slice_after`. It'll collect bytes up to and including the first terminal byte. At that point it'll put everything it has collected into an array. That array represents the first character. This process repeats until we've grabbed the bytes for every character. We finalize this with `to_a` so that we have an array of

byte arrays each of which represents one character.

We'll use a placeholder and swap its bytes just like last time. The difference here is that we'll need to calculate the number of bytes needed and build a string with that many bytes. We'll do this by using a single byte character, in this case a space, and repeating it until we have the right number of bytes. After that the method follows the same pattern we used for ASCII-2B.

The implementation for ASCII-VB is harder and a bit slower than ASCII-2B. Given everything we get in return, it's a fair trade. In 2008, a system a bit like ASCII-VB took the mantle of most used website encoding from ASCII. When Ruby 2.0.0 was released in 2013, this encoding beat ASCII again to become the Ruby default. Maybe you've heard of it.
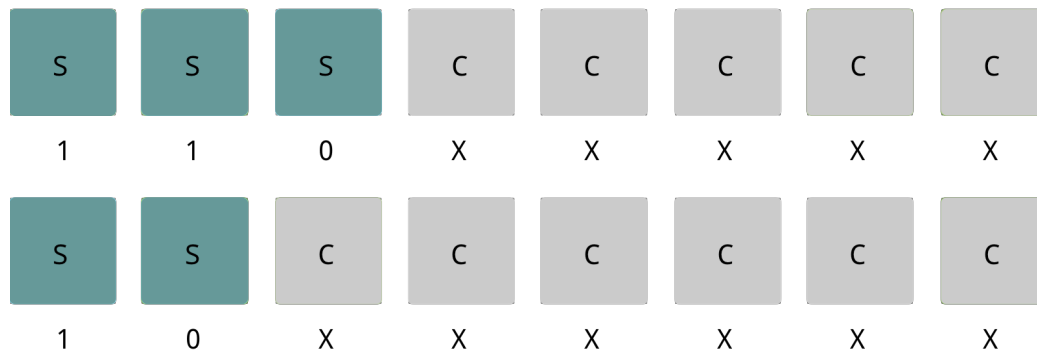
UTF-8

The Unicode Transformation Format - 8 bit (UTF-8) is the most popular encoding in the world. UTF-8 was first presented by Ken Thompson and Rob Pike at the USENIX conference in 1993. If those names aren't familiar to you, they've done a lot of great work. One of their more recent accomplishments was the invention of the Go programming language (along with Robert Griesemer).

UTF-8 was designed to fix the same problems we did with ASCII-VB. It's backwards compatible but instead of limiting it to ASCII, UTF-8 is also ISO-8859-1 (Latin-1) compatible. It also uses a different approach to indicate multi-byte characters.
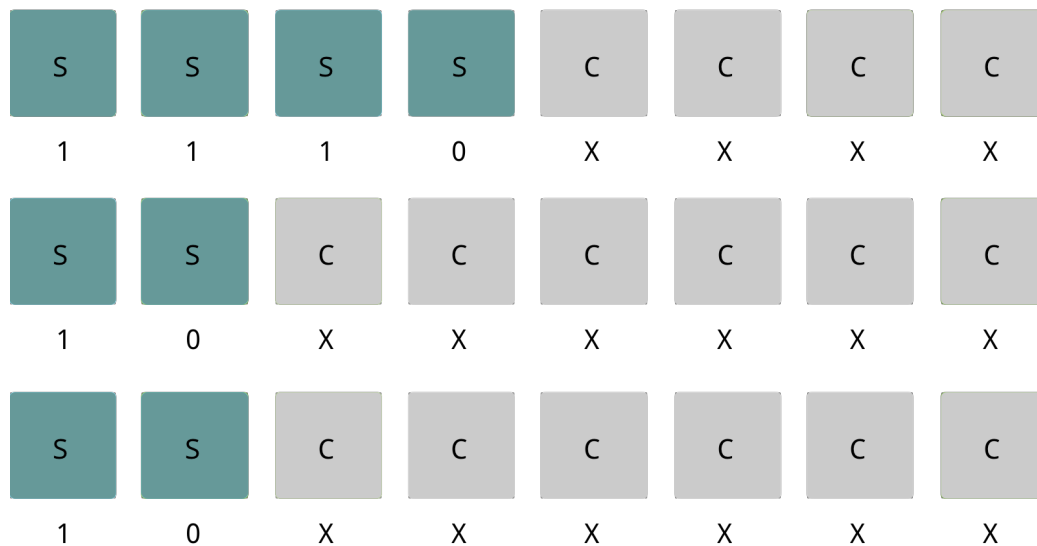
When UTF-8 has a character that's more than one byte it begins the first byte with a series of `1` bits followed by a single `0` bit. The number of leading `1` bits indicates the total number of bytes used for the character. The first byte of a two byte character would start with `110`. The first byte of a three byte character would start with

`1110`. The first byte is called the "leading byte". Bytes that follow are "continuation bytes". Each continuation byte starts with `10`. All of the other bits in the leading and continuation bytes are used to store the character.

Here we see a 2-byte character where the bits used in the UTF-8 structure are identified with "S" and the remaining character bits are identified with "C":

S	S	S	C	C	C	C	C
1	1	0	X	X	X	X	X

S	S	C	C	C	C	C	C
1	0	X	X	X	X	X	X

A three byte character leads with "1110" and has two continuation bytes:

S	S	S	S	C	C	C	C
1	1	1	0	X	X	X	X

S	S	C	C	C	C	C	C
1	0	X	X	X	X	X	X

S	S	C	C	C	C	C	C
1	0	X	X	X	X	X	X

At first you might compare this to ASCII-VB and think UTF-8 is wasteful. These changes come with advantages over what we've done. Having the total number of

bytes for the character encoded into the leading byte proves to be quite useful. It allows code to skip from one character to the next without having to read the bytes in between. Even better, it means that you can validate characters. If we were reading ASCII-VB and hit a byte with a leading `1` followed by a byte with a leading `0` we'd assume that was the entire character. In truth there may have been a third byte that preceded the others. That third byte could have been accidentally removed or lost in transmission. UTF-8 can be checked for validity because we know the number of bytes needed and we know each continuation byte must start with `10`. If we begin reading and the first byte starts with `10` then we know that something has gone wrong.

The strong bit structure underlying UTF-8 also makes it fairly easy to guess with heuristics. If you receive a document with no encoding listed (e.g. a web page), the application reading it could make a pretty good guess about whether it's UTF-8 or not.

UTF-8 is space efficient (for most use cases), widely supported, and is the default for Ruby. If you're going to use something other than UTF-8, you'd better have a great reason.

Let's spend some time digging into UTf-8 and see what it has to offer.

5 | Character Sets & Code Points

Character sets and code points are abstractions that sit between bytes and encodings. A character set defines a group of characters, their order, and it assigns each an identifier. The identifier is known as a "code point". It allows for character interaction without having to understand the underlying byte structure of a character.

They're often confused but UTF-8 and Unicode are two different things. Unicode is a standard that includes a character set and bunch of rules about how those characters interact. UTF-8 is an encoding that implements the Unicode character set.

Abstractions are distractions unless they add value. Character sets and code points create two valuable benefits.

If you were to make a totally new encoding with no concern for backwards compatibility would you choose fixed-width or variable-width characters? Each approach comes with advantages and disadvantages. By using a character set you can provide both implementations. The user can choose the implementation that best addresses their needs. If they receive characters in the other format they can easily transform them by referencing the code points used in both implementations.

Anything that builds on top of this abstraction immediately works for both implementations. Fonts are built to map glyphs to characters based on the character set. A Unicode font will work with any implementation instead of only UTF-8. Methods

like `upcase` can be written so that they'll function against any Unicode implementation.

Picking the best implementation for a situation and still getting lots of generic tools to work with it is a big win.

Not every encoding was built with character sets in mind. Take US-ASCII and ISO-8859-1 for example. In cases like that, the value of the byte representing the character *is* the code point. They essentially have a defacto character set based on their implementation.

Unicode

Unicode is the most popular character set in the world. It contains around 137,000 characters and each year more characters are added. It even contains a wide variety of emoji 😊.

The Unicode character set has been implemented in a variety of Unicode Transformation Formats (UTF). Each UTF is followed by a number that indicates the size of the code unit. That's the smallest number of bits that can be used or incremented by. In the case of UTF-8 that means the smallest amount of bits you can use to represent a character is 8. If you need more bits, they have to be added in groups of 8.

UTF-16 is an implementation that requires at least 16 bits. That doesn't turn out to be enough bits to store every possible character. The higher characters are implemented by adding another 16 bits. That's the minimal unit of increment. UTF-16 has all of the negatives of UTF-8 while also failing to retain backwards compatibility with ASCII. Its one common use involves some Asian languages. For some languages their characters require 3 code units in UTF-8 (i.e. 24 bits) but only one

code unit in UTF-16 (i.e. 16 bits). As a result storing strings using UTF-16 saves space.

There's also UTF-32 which uses 32-bit increments. However, you only need 32 bits to store all of the Unicode characters. No incrementing needed. UTF-32 therefore has the advantages that come with being fixed-width.

Each of these is part of the Unicode Standard which is maintained by the Unicode Consortium. It's a non-profit organization with certain large tech companies acting as due-paying members. Paying dues makes you a voting member with influence over the direction of the standard. They do however accept new character proposals from anyone. The Unicode Consortium has been around publishing guidelines for Unicode since 1991.

Code Points

Code points are built into the byte structure of the characters. For UTF-8 characters, the code point is what you get if you read the non-structural bits. As an example, let's break down the letter "à". It has a code point of 224 and uses two bytes.

S	S	S	1024	512	256	128	64
1	1	0	0	0	0	1	1

S	S	32	16	8	4	2	1
1	0	1	0	0	0	0	0
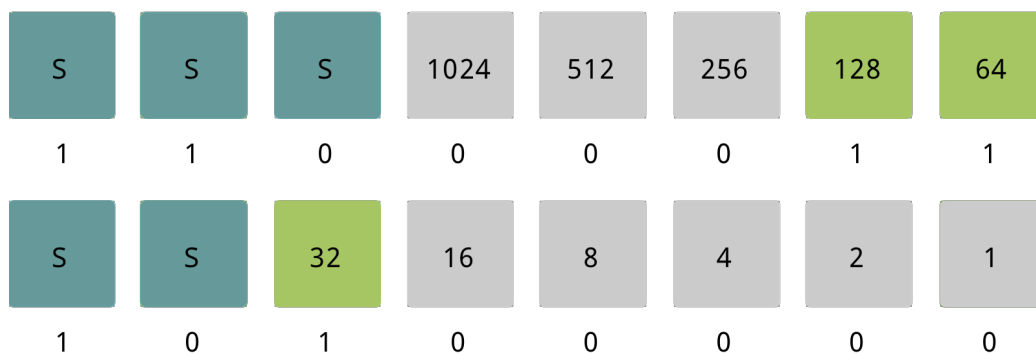
In the non-structural bits we have 128, 64, and 32 turned on. These add up to our code point of 224. UTF-32 is fixed-width so it has no need for structural bits. The

code point is literally the value of the bits.

~2B	~1B	~536M	~268M	~134M	~67M	~33M	~16M
0	0	0	0	0	0	0	0

~8M	~4M	~2M	~1M	~524k	~262k	~131k	65536
0	0	0	0	0	0	0	0

32768	16384	8192	4096	2048	1024	512	256
0	0	0	0	0	0	0	0

128	64	32	16	8	4	2	1
1	1	1	0	0	0	0	0

We can extract the code points in a string with the `codepoints` method.

```
>> 'àbç'.chars
=> ["à", "b", "ç"]

>> 'àbç'.codepoints
=> [224, 98, 231]

>> 'àbç'.bytes
=> [195, 160, 98, 195, 167]
```

We can confirm that they're shared between UTF implementations.

```
>> 'àbç'.encode('UTF-8').codepoints
=> [224, 98, 231]
```

```
>> 'àbç'.encode('UTF-32BE').codepoints # We'll explore "BE"
    later.
=> [224, 98, 231]
```

It's important to remember that these aren't completely universal. Different character sets will choose different code points to represent their characters. If two character sets do share the same numbers it's likely to be a result of both of them extending ASCII.

```
>> 'àbç'.codepoints # UTF-8
=> [224, 98, 231]

>> 'àbç'.encode('IBM437').codepoints
=> [133, 98, 135]
```

We can see that UTF-8 and IBM437 share the same code point for "b" but differ when it comes to "à" and "ç". This is an example of where they share an ASCII base and then diverge.

Code points can also be accessed with `each_codepoint`.

```
>> 'àbç'.each_codepoint { |c| puts c }
224
98
231
=> "àbç"
```

Adding a character by code point to a string can be done with the << operator (or `concat`). All you have to do is provide the integer code point instead of a string.

```
>> a = 'àbç'
=> "àbç"
>> a << 100 # 'd'
=> "àbçd"
>> a
=> "àbçd"
```

Code points are a great way to look up a character that won't display. They're also more portable than working with bytes. If you start writing code that uses byte methods stop and consider code points.

Recap

UTF-8 is the most popular encoding built on Unicode, the most popular character set. It's the most generally useful of the Unicode encodings but can be easily converted to a different Unicode implementation when needed. This is because of the shared Unicode character set and the code points it declares. While UTF-8 is almost always what you want, it's not always what you get. Sometimes you'll need to work with other encodings. Let's see what Ruby has to offer in the way of options.

6 | Ruby Encodings

There are a lot of encodings in the wild. The Internet Assigned Numbers Authority has deemed 257 encodings as acceptable for the internet. You're likely to duck most of them even over a long and illustrious career. Ruby also ducks many of them but does manage to support 101 encodings. Which is plenty. Unless it's missing the one you need. Each encoding is represented as an instance of the `Encoding` class.

```
>> Encoding.list
=> [#<Encoding:ASCII-8BIT>, #<Encoding:UTF-8>, #<Encoding:US-
   ASCII>, ...]
```

Every encoding has a name. Some however, live like international spies. They have a number of aliases that they go by. These aliases can occasionally help clarify the encodings. In the list above there's "ASCII-8BIT" and "US-ASCII". Why are there two ASCII encodings? Aliases will help us sort this out.

```
>> Encoding.aliases
=> {"BINARY"=>"ASCII-8BIT", "ASCII"=>"US-ASCII", ...}
```

The "US-ASCII" encoding is the ASCII we're familiar with. The other, "ASCII-8BIT" represents binary data that's being stored in a string. These aliases can be used in place of the canonical name anywhere you might need it. For example, we can grab any encoding we want with the `find` method.

```
>> Encoding.find('BINARY')
```

```
=> #<Encoding:ASCII-8BIT>
```

Each encoding is also tied to a constant within the `Encoding` namespace. All we have to do is substitute underscores for dashes and we're ready to go. Like before, we can do this same thing for any alias we want.

```
>> Encoding::ASCII_8BIT
=> #<Encoding:ASCII-8BIT>

>> Encoding::BINARY
=> #<Encoding:ASCII-8BIT>
```

The canonical name for an encoding can always be found by calling `name`. Aliases, like the ones found in `Encoding.aliases`, can be found by calling, you guessed it, `names`. That wasn't your guess? In fairness, there's no method to get the aliases from the instance of an encoding. We can use `names` to get the aliases *and* the canonical name. If we really want the aliases we can use `names` and remove the canonical name.

```
>> e = Encoding.find('BINARY')
=> #<Encoding:ASCII-8BIT>
>> name = e.name
=> "ASCII-8BIT"
>> names = e.names
=> ["ASCII-8BIT", "BINARY"]
>> aliases = e.names - [e.name]
=> ["BINARY"]
```

Et voilà.

Dummies

Some encodings are dummies. Dummy encodings are incomplete implementations that must be used with caution. The good news is they're honest dummies. They'll out themselves if you call `dummy?`.

```
>> Encoding.list.select(&:dummy?)
=> [
 #<Encoding:UTF-16 (dummy)>,
 #<Encoding:UTF-32 (dummy)>,
 #<Encoding:IBM037 (dummy)>,
 #<Encoding:ISO-2022-JP (dummy)>,
 #<Encoding:ISO-2022-JP-2 (dummy)>,
 #<Encoding:CP50220 (dummy)>,
 #<Encoding:CP50221 (dummy)>,
 #<Encoding:UTF-7 (dummy)>,
 #<Encoding:ISO-2022-JP-KDDI (dummy)>
 ]
```

All of the current dummy encodings are either not US-ASCII compatible or they're byte-order dependent. An example of an incompatible encoding is IBM037. It has all the same characters as ISO-8859-1 but the layout is entirely different. Instead of "a" having a code point of 97 it's 129. One downside of dummies is that they're only partially implemented. That means some methods simply fail when called.

```
>> 'a'.encode('IBM037').upcase
Encoding::CompatibilityError: incompatible encoding with this
    operation: IBM037
```

Encodings like the ISO-2022 and CP5022x series are stateful encodings. They contain multiple character sets that can be switched between by using special escape characters. With ISO-2022-JP we can switch between a set of Japanese characters and US-ASCII. Implementing stateful encodings is difficult work. Consider what it would take to insert a character into the middle of a string. The code would have to

trace back to figure out what character set was being used. If it's the wrong character set it needs to include the escape sequences to switch to ASCII and then back out. Deleting that newly inserted character would have to undo the process. Problems like this make them troublesome to implement.

UTF-7 makes an appearance on the list because you can't implement it in an 8-bit system. So, why have it at all? It turns out some email systems still use it. In fact `Net::IMAP` contains `encode_utf7` and `decode_utf7` methods. They produce `ASCII-8BIT` (i.e. `BINARY`) output though.

This leaves us with UTF-16 and UTF-32. These are common encodings and Ruby does actually support them. They show up on the dummy list because they're both byte-order dependent.

Early computers may have settled in on 8 bit bytes but they never did agree on what order to store them in. Some systems start a byte sequence with the most significant byte. These systems are referred to as big-endian. Systems that start with the least significant byte are little-endian.

The term "endian" may seem odd and it is. Danny Cohen coined the term in a message he wrote about the conflict between those who preferred the most significant byte first vs those who preferred the least significant byte first [1]. In it he referred to a passage from *Gulliver's Travels* in which citizens dispute the proper end by which to open an egg. There were those who would open it from the big end which were "Big-Endians". The opposition opened eggs from the little end and were "Little-Endians". Cohen's terms were meant to remark on the zealous nature of the debate but ended up becoming the official descriptors.

The recommendation by the Unicode Consortium is to provide one encoding for big endian, one for little endian, and one to rule them all.

```
>> Encoding.name_list.grep(/UTF-16/)
```

---

[1] https://www.ietf.org/rfc/ien/ien137.txt

```
=> ["UTF-16BE", "UTF-16LE", "UTF-16"]


>> Encoding.name_list.grep(/UTF-32/)
=> ["UTF-32BE", "UTF-32LE", "UTF-32"]
```

In UTF-32 the code point is encoded directly into the bytes with no structural bytes. The character "語" has the code point 35,486. Converted to binary and stored across four big-endian bytes it would be "00000000 00000000 10001010 10011110" or "0 0 138 158" using decimals. Let's see how the bytes are laid out in both the big-endian and little-endian encodings.

```
>> '語'.encode('UTF-32BE').bytes
=> [0, 0, 138, 158]


>> '語'.encode('UTF-32LE').bytes
=> [158, 138, 0, 0]
```

What do we get if we use the non-specific UTF-32 encoding?

```
>> '語'.encode('UTF-32').bytes
=> [0, 0, 254, 255, 0, 0, 138, 158]
```

It adds an extra byte before continuing on with big-endian. That extra byte at the beginning is a byte order mark (BOM). The BOM tells a system how to interpret the ordering of the bytes. This way big-endian and little-endian systems can share data without messing each other up.

The BOM is an actual Unicode character with the hexadecimal code point FEFF. We can see in the example above that it's stored as `[0, 0, 254, 255]` in UTF-32BE. If our code tries to use UTF-32LE it'll read the byte backwards and get FFFE0000. That code point is above the maximum allowable Unicode code point. The system can then try again with UTF-32BE. It'll read the BOM correctly and know that the data is big-endian encoded. Once we know the endianness we can easily swap it. This same process would hold true for a stream of little-endian UTF-32 being read

as big-endian UTF-32.

With UTF-16 the reverse of FEFF is FFFE which does not exceed the highest possible code point. Unicode solves this by reserving FFFE as a non-character. If you read this non-character then you know something has gone wrong.

Finally, UTF-8 is always big-endian but it can still include a BOM. It's not common but it can happen. When it does the BOM doesn't indicate the byte order it simply helps to identify the data stream as UTF-8.

US-ASCII Compatible

Lots of encodings are supersets of US-ASCII. They've extended the base implementation to add their own characters. You can see if an encoding does this by checking to see if it is US-ASCII compatible. Ruby provides the class method `ascii_compatible?` to check for compatibility.

```
>> Encoding::UTF_8.ascii_compatible?
=> true

>> Encoding::ISO_8859_1.ascii_compatible?
=> true

>> Encoding::UTF_32.ascii_compatible?
=> false
```

This compatibility only says that the encoding may contain US-ASCII characters. It doesn't tell us if a specific string can be safely converted to US-ASCII. By that, I mean that the string contains only US-ASCII characters. For that we'd use `ascii_only?`.

```
>> 'abc'.ascii_only?
=> true
```

```
>> 'àbç'.ascii_only?
=> false
```

Script Encodings

Ruby scripts will default all strings to UTF-8. We can change that by adding a special comment, called a pragma, to the top of the script. The comment must have the word "coding" or "encoding" followed by a colon, a space, and the name of the encoding you want.

```
# encoding: ASCII
```

This needs to be the very first line of the file. If your file has a shebang then it needs to be the first line after the shebang.

```
#!/usr/bin/env ruby
# encoding: ASCII
```

You can do this with any Ruby file but I wouldn't use it for anything but scripts. If your application needs a string that's not UTF-8, you're better off making it explicitly in the place it's needed.

We can determine the current encoding of a file by using the magical __ENCODING__ variable.

```
# encoding: ASCII

puts __ENCODING__.inspect
```

Putting that into a file and running it outputs "US-ASCII". Remember, "ASCII" is an alias for "US-ASCII" so that's what actually gets set.

The `__ENCODING__` variable also works inside IRB sessions. We can set the encoding for IRB with `--encoding` or its shorter equivalent `-E`.

```
$ irb -E ASCII
>> __ENCODING__
=> #<Encoding:US-ASCII>
```

How Many Encoding Settings Are There?

Ruby has four separate settings for selecting encodings. We can set the locale, filesystem, internal, and external encodings. That's a lot to consider. Thankfully, you won't have to deal with them too often.

The locale should reflect your operating system locale. On a *nix system you can run `locale` for information about how you're set up.

```
$ locale
LANG="en_US.UTF-8"
LC_COLLATE="en_US.UTF-8"
LC_CTYPE="en_US.UTF-8"
LC_MESSAGES="en_US.UTF-8"
LC_MONETARY="en_US.UTF-8"
LC_NUMERIC="en_US.UTF-8"
LC_TIME="en_US.UTF-8"
LC_ALL=
```

Ruby will use `LANG` to determine your system locale. You can play with this by changing `LANG` to something different when running IRB. Ruby aliases this encoding as `'locale'` so it can be easily found.

```
$ LANG=en_US.US-ASCII irb
>> Encoding.find('locale')
=> #<Encoding:US-ASCII>
```

Changing LANG will also adjust the filesystem, internal, and external encodings.

Next up is the filesystem encoding. It describes the encoding used to store path and file names. Once again Ruby provides a `'filesystem'` encoding alias.

```
>> Encoding.find('filesystem')
=> #<Encoding:UTF-8>
```

My system is UTF-8 but that won't always be the case. It's possible that a filesystem could use UTF-16. The good news is that Ruby will convert strings for us.

```
>> require 'fileutils'
=> true
>> file_name = 'ğ.txt'.encode('ISO-8859-3')
=> "\xBB.txt"
>> FileUtils.touch(file_name)
=> ["\xBB.txt"]
```

Here we've written an ISO-8859-3 encoded file name to a UTF-8 filesystem. Listing the files in the directory displays a perfectly legible "ğ.txt".

The final two, internal and external, are used for input and output (IO) operations. The external encoding represents the encoding of the source being read into the system. The internal encoding is the encoding that you would like the source to be converted into. You can get to these by finding the `'internal'` and `'external'` encodings.

```
>> Encoding.find('external')
=> #<Encoding:UTF-8>
>> Encoding.find('internal')
=> nil
```

Here we see that data is expected to be UTF-8 and will be read in as `nil`. For the internal encoding, `nil` means that the data will not undergo any transformation.

You can change these values with the `-E` flag when running `ruby` or by setting

`Encoding.default_external=` and `Encoding.default_internal=`. I would recommend against it though. Those are global changes. Instead, we can explicitly call out these expectations when reading IO.

The `IO` class (and by extension the `File` class) provides a means to set these encodings on a per-use basis. The `open` method takes a file location and a mode of operation. The mode explains to Ruby how you'd like to interact with the file. For example, setting the mode to `'r'` makes the file read only. That portion of the mode is called the fmode. It specifies how to read and/or write to the file. After the fmode we can specify the external encoding and optionally the internal encoding. We do this by separating each part with a colon.

```
File.open('file.txt', 'r:ISO-8859-1:UTF-8')
```

This opens `file.txt` as a read-only file whose contents are ISO-8859-1 but should be converted to UTF-8. If you want to use whatever Ruby has declared as the default you can replace the name of the encoding with a –.

```
# use default external but convert it to ASCII
File.open('file.txt', 'r:-:US-ASCII')
```

I find the colon separated version a bit hard to read. Fortunately, we can also pass the encodings as options.

```
File.open('file.txt', 'r',
  external_encoding: 'US-ASCII',
  internal_encoding: 'UTF-8'
)
```

If they're the same we can replace `:external_encoding` and `:internal_encoding` with `:encoding`. These options also accept the use of – to denote the default. Passing `nil` to `:internal_encoding` indicates that no transformation needs to happen.

Ruby does its best but not all characters can be mapped from one encoding to an-

other. When it can't transform a character, it'll let us know.

```
>> File.open('file.txt', 'w',
..    external_encoding: 'US-ASCII'
.. ) do |f|
..    f.puts 'abc'
..    f.puts 'ğ'
..    f.puts 'def'
.. end
Encoding::UndefinedConversionError: U+011F from UTF-8 to US-
    ASCII
```

This doesn't stop Ruby from writing the file or adding "abc" to it. Everything prior to the error still happened. We'll look at fixing encoding errors in chapter 13. In the meantime, know that there are ways to ensure Ruby doesn't stop like this.

Recap

We've figured out how to find encodings in Ruby and how to spot a dummy. We know all the ways to set the encoding and how to handle input and output when the encodings differ from our default. Now that we're a bit more comfortable with encodings, let's go back to UTF-8. When you create an encoding to handle every language in the world, there are bound to be some edge cases. We're going to take a side trip and discuss one very important edge case.

7 | Normalization Forms

How can we have two strings that look the same but aren't equal?

```
>> a
=> "é"
>> b
=> "é"
>> a == b
=> false
```

Maybe they have different encodings?

```
>> a.encoding
=> #<Encoding:UTF-8>
>> b.encoding
=> #<Encoding:UTF-8>
```

Nope. Time for a deeper look. Let's check their code points.

```
>> a.codepoints
=> [233]
>> b.codepoints
=> [101, 769]
```

For `a` we have a single code point, 233, that prints "é". With `b` we have a pair of code points. The first code point, 101, is the standard "e" character. The second, 769, is a combining acute accent mark that displays as "◌́". When Ruby runs across a

character followed by a combining character it renders a combination of the two. The result is a visually identical but different "é".

The Unicode Standard supports both of these approaches. It also lays out text normalization rules for converting between the two. Text normalization comes in a few forms that are distinguished by two traits. Characters can be composed or decomposed. The single 233 code point is the composed version of the character. The decomposed version, consists of the pair of code points which are visually combined. The other trait is canonical vs compatible equivalence. In total this provides for four distinct normalization forms.

Canonical Equivalence

> *Canonical equivalence is a fundamental equivalency between characters or sequences of characters which represent the same abstract character, and which when correctly displayed should always have the same visual appearance and behavior. - Unicode Standard Annex #15*[1]

The two versions of "é" described above would be considered canonically equivalent. Both produce a character with the same appearance and behavior. To the eye they're identical.

The composed single code point is the Normalization Form Canonical Composition (NFC) version of the character. The decomposed pair is the Normalization Form Canonical Decomposition (NFD) form of the character. To switch between these normalization forms, we can use `unicode_normalize` or its mutating companion `unicode_normalize!`.

```
>> "é".codepoints
```

---

[1]http://unicode.org/reports/tr15/#Canon_Compat_Equivalence

```
=> [233]
>> "é".unicode_normalize(:nfd).codepoints
=> [101, 769]

>> e = '' << 101 << 769
=> "é"
>> e.codepoints
=> [101, 769]
>> e.unicode_normalize(:nfc).codepoints
=> [233]
```

At first glance this transformation may appear to be a safe one. With "é" that's true. We can switch between the two fluidly with no loss of information. That isn't always the case.

As a rule, before running a composition, the character is first decomposed and then composed. When we run `unicode_normalize(:nfc)` on the composed "é", it is first ripped apart and then re-combined. For some characters this separation and subsequent combination results in a loss of information. These are special Unicode characters called "singletons".

Consider "Å" which is the Latin capital letter "A" with a ring above it. It's the same symbol that's used for the Ångström unit of length. Unicode has both characters and they're given separate code points. The problem is that they both decompose the same way.

```
>> latin_letter = '' << 197
=> "Å"
>> angstrom = '' << 8491
=> "Å"

>> latin_letter.unicode_normalize(:nfd).codepoints
=> [65, 778]
>> angstrom.unicode_normalize(:nfd).codepoints
=> [65, 778]
```

Both decompose into "A" and "◌̊". If we compose these two characters are we trying to get the Latin letter or unit symbol for an Ångström? The transformation can't divine our intent so it always transforms it into the Latin letter. Transforms on the Ångström character will result in a visually identical but different character.

```
>> angstrom = '' << 8491
=> "Å"

>> angstrom.unicode_normalize(:nfc).codepoints
=> [197]
```

Singletons occur for any group of characters represented by the same abstract character. Take the Greek letter omega "Ω". The same abstract character is used for the ohm unit of electrical resistance. Again, Unicode provides different code points but even without a combining mark the NFC transformation is destructive.

```
>> omega = '' << 937
=> "Ω"
>> ohm = '' << 8486
=> "Ω"

>> omega.unicode_normalize(:nfc).codepoints
=> [937]
>> ohm.unicode_normalize(:nfc).codepoints
=> [937]
```

Being visually identical masks the damaging change in meaning. Screen readers may choose to pronounce the two characters differently. It also changes the behavior of sorting and searching for these characters. This change might be good if you're aware that it's happening. Users may be searching with the wrong character by accident or your data may have gotten the wrong character to start with. In those cases, converting omega and ohm to the same character might be advantageous.

Searching over accented words might be causing issues. Consider searching for the word "resume" and missing a result because it used "résumé". Transforming to NFD

and stripping the range of combining accents could provide a more searchable text.

```
>> COMBINING_MARKS = "\u0300-\u036f"
>> 'résumé'.unicode_normalize(:nfd).delete(COMBINING_MARKS)
=> "resume"
```

We've split the "é" apart and then deleted the combining accent. Now we can do the same to search terms and provide better results. This may also let us sort a field in a more natural way. When doing something like this, remember to always retain a copy of the original text.

Stripping our combining marks leads us to another issue. Ruby is treating them like characters. Otherwise there's no way we could have removed them with `delete`. What that means is we may end up with some confusing results if we don't know the string is NFD.

```
>> "é".unicode_normalize(:nfd).size
=> 2
```

What looks like one character is actually two so we get a string size we wouldn't expect. Ruby will always treat each character independently.

```
>> 'résumé'.unicode_normalize(:nfd).split('e')
=> ["r", "́sum", "́"]
```

Look closely at the leading quote for the second and third array items. The combining mark from each "e" remains and gets visually merged into the quote preceding it. The NFC version doesn't contain an "e" and is never split apart.

```
>> 'résumé'.split('e')
=> ["résumé"]
```

NFD characters can be useful but they can also wreak a lot of havoc if you don't know what's going on.

Compatibility Equivalence

> *Compatibility equivalence is a weaker type of equivalence between characters or sequences of characters which represent the same abstract character (or sequence of abstract characters), but which may have distinct visual appearances or behaviors. - Unicode Standard Annex #15[2]*

NFC and NFD both represent the canonical form of normalization. The other two forms are Normalization Form Compatibility Decomposition (NFKD) and Normalization Form Compatibility Composition (NFKC). These forms represent transformations that aren't exactly identical but they're close enough. Kind of like using a "K" to represent "Compatibility".

Compatibility transformations often result in a simpler representation of the characters. Here are a few transformations to give you an idea of what I mean.

```
>> '𝒜'.unicode_normalize(:nfkd)
=> "A"
>> '½'.unicode_normalize(:nfkd)
=> "1⁄2" # 3 characters with a "fraction slash"
```

It's possible for this to result in changes that appear to be errors.

```
>> '3²'.unicode_normalize(:nfkd)
=> "32"
```

Our text which contained 3 to the power of 2 now reads as "32". Oops. One useful case is the ability to break down ligatures. Ligatures are series of letters which are pushed together to create a single character. One example is "ﬃ" which combines "ffi".

---

[2]http://unicode.org/reports/tr15/#Canon_Compat_Equivalence

```
>> 'office'.unicode_normalize(:nfkd)
=> "office"
```

NFKC, like NFC, first decomposes then composes. Letters like "ö" will recompose while ligatures like "ffi" won't. Similar to the singletons, the transform can't know if you want the ligature or not.

```
>> office = 'öffice'
=> "öffice"

>> office.codepoints
=> [246, 64259, 99, 101]
>> nfkc = office.unicode_normalize(:nfkc)
=> "öffice"
>> nfkc.codepoints
=> [246, 102, 102, 105, 99, 101]
```

Like before, transforming text in this way can help to normalize it for tasks like sorting and searching. The compatibility form is much more disruptive which may be a good thing depending on your needs.

Recap

NFD will break characters down into their constituents so long as the visual output remains unchanged. NFC puts those parts back together the best it can. Throw a "K" in (NFKD and NFKC) and you get "Compatibility" mode which is more liberal in its replacements. Transformations should be thought of as destructive, so always keep a copy of the original.

If we're not sure what we have, we can check using `unicode_normalized?`. By default it'll check for NFC normalization but we can pass it a symbol to check the other types.

```
>> 'résumé'.unicode_normalized?
=> true
>> 'résumé'.unicode_normalize(:nfd).unicode_normalized?
=> false
>> 'résumé'.unicode_normalize(:nfd).unicode_normalized?(:nfd)
=> true
```

This check only works when the string maintains a consistent normalization form. In this example, the first "é" in "résumé" is a single composed character while the second is decomposed.

```
>> resume
=> "résumé"
>> resume.codepoints
=> [114, 233, 115, 117, 109, 101, 769]

>> resume.unicode_normalized?(:nfc)
=> false
>> resume.unicode_normalized?(:nfd)
=> false
```

At this point we've looked at bits, bytes, character sets, code points, and the ins and outs of encodings. Armed with this knowledge let's start creating some strings.

8 | Creation

No class in Ruby provides as many creation options as `String`. There are single quotes, double quotes, a question mark, percent signs, heredocs, and good ole `new`. Each constructor has its own use case. Each comes with its own nuances.

Raw or Processed

The most straight-forward constructor is the single quote. It creates what I call "raw strings". With these raw strings, what you type is what you get. The only deviation to that rule is when you have a single quote in the middle of a string and have to escape it with a back slash.

```
>> puts 'Aaron\'s string\nis great!'
Aaron's string\nis great!
```

Had we used double quotes for that, the `\n` would have created a new line instead of literally displaying a slash followed by an "n".

```
>> puts "Aaron's string\nis great!"
Aaron's string
is great!
```

Single quotes couldn't care less about what we want. Their purpose is to mirror

back what they're given. If single quotes create raw strings, then double quotes create processed strings. Double-quoted strings interpret escape codes and allow for interpolation. That's when we jam code right into the middle of a string.

Interpolation is the process of executing code within a string and replacing it with the result. Inside the string we'll use `#{}` with code that we want to run placed between the curly braces.

```
>> name = 'Aaron'
=> "Aaron"
>> "Hello #{name}!"
=> "Hello Aaron!"
```

In this case our result was a string but any non-string result will have `to_s` called on it.

```
>> "PI is about #{Math::PI.round(2)}"
=> "PI is about 3.14"
```

There are three instances when you can leave off the curly braces. Global, class, and instance variables can be interpolated into a string with only a `#` preceding them.

```
>> $word = 'don\'t'
=> "don't"
>> @@word = 'do'
=> "do"
>> @word = 'this'
=> "this"
>> "#$word #@@word #@word"
=> "don't do this"
```

You may have guessed from the example that I'm not a fan of doing this. I find that the curly braces increase the visibility of the interpolated section. It also breaks down the second we decide to do anything, like call `capitalize` on the first variable.

The other powerful feature of double quotes is the interpreting of escape sequences. These are commands issued within a string that start with a \. Earlier we saw \n (i.e. newline) which is an escape sequence that moves the output cursor down a line before continuing. It's a lot better than having to manually add a return.

```
>> puts ['a', 'b', 'c'].join('
.. ')
a
b
c
=> nil

>> puts ['a', 'b', 'c'].join("\n")
a
b
c
=> nil
```

That first one is pretty gross. I wouldn't want to see that in the middle of my code. It's also impossible to indent properly. Ruby provides lots of escape sequences that make life easier.

Escaping

Some escape sequences date back to the US-ASCII days. As a result, a number of them are rather outdated. They were built when most output was sent to a printer rather than a monitor. The newline character, also known as "line feed", would move the paper down a line and then carriage return (i.e. \r) would move the print head back to the start of the line.

Some computers stuck with the original behavior while other reworked the newline to automatically return to the beginning of the line. If you've ever had to convert

line ending between Windows and *nix systems, you've seen the fallout from this.

The mighty carriage return remained unchanged through this process. It still moves the output cursor to the start of the line.

```
>> puts "hello\rc"
cello
```

What happened here is Ruby printed the word "hello", then moved the output cursor to the start of the line, and then printed a "c". That "c" overwrote the "h" and we got "cello".

We can see a practical use of carriage returns by creating a progress bar for the terminal.

```
1.upto(100) do |percent|
  print '['
  print ('=' * (percent / 5)).ljust(20)
  print ']'
  print " #{percent.to_s.rjust(3)}%\r"

  percent == 100 ? puts : sleep(0.05)
end
```

If you paste that into IRB you'll get a result that looks like this:

```
[==============        ]  72%
```

The progress bar will slowly fill until it hits 100% complete. The `\r` at the end of the final `print` resets the output to the beginning of the line. By using `ljust` and `rjust` we can ensure the same amount of characters are output with each loop. These methods, along with `center`, pad and justify content.

```
>> 'a'.center(5)
=> "  a  "

>> 'a'.ljust(5)
```

```
=> "a    "

>> 'a'.rjust(5)
=> "    a"
```

All three accept a second argument that we can use to specify a pattern to pad with.

```
>> 'a'.ljust(5, '*')
=> "a****"

>> 'a'.ljust(5, '*=@')
=> "a*=@*"
```

Beyond newlines and carriage returns we can also use form feed (i.e. \f) and vertical tab (i.e. \v) to move the output cursor. Each moves the cursor down a line without reseting it to the beginning of the line.

```
>> puts "a\vb"
a
 b
>> puts "c    \fd"
c
     d
```

They work the same now but they used to have different meanings. The form feed would move to the next page. It acted like a page break. Vertical tabs could be used to align content on the page when used with printers that had specialized tab belts to jump down to specific sections on the paper. The same behavior existed for horizontal tabs (i.e. \t). Today, we use horizontal tabs to argue about how to space code.

Code spacing aside, horizontal tabs move the output cursor to the next available tab slot. A tab slot is a pre-defined distance from the start of the line. Horizontal tabs typically occur every 8 characters, though the distance is adjustable based on the system.

```
>> puts "a\tb\tc" "\n" "d\te\tf" "\n" "hijklmnopq\tr"
a       b       c
d       e       f
hijklmnopq      r
```

The `\t` will always look for the next tab location. We can see how "hijklmnopq" barreled through the second tab slot and forced "r" into the third slot. This can make it difficult to present tabular data with tabs. Ironic, no?

Another relic is the backspace (i.e. `\b`) character. Unlike the way we often think of it, backspace does not delete the last character, it only moves the cursor back one character.

```
>> puts "care\b\bk"
cake
```

Here we printed "care", moved the cursor back two characters, and then overwrote the "r" with a "k".

One old but fun escape sequence is the bell (i.e. `\a`). It used to ring a literal bell on the receiving end. The bell alerted operators that a message was coming. With modern shells it's sometimes used to signal completion of a task or that you've made a mistake. You can try it out for yourself with `puts "\a"`.

Characters by Code Point

There are several ways to add a character by its code point. We've used << to do it.

```
>> '' << 65
=> "A"
```

That's quick, and dirty, and useful in IRB but there are better ways. Let's go over

some options. With `\nnn` we can use the octal version of the code point to add the character. If we know the hexadecimal version there's `\xnn`.

```
>> "\101\x41"
=> "AA"
```

Both of these work but I don't like them. The hexadecimal version leads with an "x" but the other gives you no hints that it's octal. They're also fixed in size. We can't use them with an octal over 999 or a hexadecimal above FF.

Unicode characters can be put into a string with `\unnnn`. It expects the number to be in hexadecimal. This is generally how Unicode code points are referenced so it makes lookup a breeze.

```
>> "\u0041"
=> "A"
```

This has many of the same problems that `\nnn` and `\xnn` have. You have to use exactly the right number of digits and can't go above FFFF. The emoji characters are higher than that. Are we going to settle for that? Nope!

Like an anime character, `\u` has a second, more powerful form. Instead of using exactly four digits we'll surround the digits with curly braces.

```
>> "\u{1f602}"
=> "😂"
```

No need to pad small numbers with zero or stop at FFFF. Additionally, the curly braces stand out well within the string.

As a general rule, I try to avoid solutions like these. If I need a "ü" I'd prefer to find the actual character rather than lean on a code point. Ruby can handle the Unicode characters, as can most editors. It also means the character is in plain sight instead of hiding behind a number.

There are times where `\u` is appropriate. It works particularly well for characters that are not visually obvious. Creating NFD accented characters, calling out specific characters which look similar to others (e.g. omega vs ohm), and pointing out special types of whitespace are all valid uses.

```
>> ['a', 'b', 'c'].join("\u{200a}") # 200a - thin space
=> "a b c"
```

I could have used the actual thin whitespace character. Of course, a monospace font isn't going to distinguish that from a normal space. Even if the font showed a difference, would you notice it? I wouldn't.

It also makes the character easier to lookup on Unicode charts. I do recommend, as I did above, describing the character to help maintain code skimmability. It'll save everyone the hassle of looking it up whenever they run across it.

A Bunch of Code Points

Using `\u` in a string is fine if we've got a point or two to add. If we have lots of code points it might be better to use `Array#pack`. It will turn an array of code points into a string based on a template we give it.

```
>> [65, 128514, 90].pack('UUU')
=> "A😂Z"
```

These three code points were converted into a string based on the `'UUU'` template. That template tells `pack` to interpret the first three entries of the array as Unicode code points. If I add a fourth item to the list it won't show up.

```
>> [65, 128514, 90, 33].pack('UUU')
=> "A😂Z"
```

With `'UUU'` we've only told `pack` to build a string based on the first three entries of the array. Listing the exact number of code points to read isn't alway practical. To repeat a portion of the template we can use a `*`. For example, `'U*'` will read as many entries as we give it.

```
>> [65, 128514, 90, 33].pack('U*')
=> "A😂Z!"
```

The `pack` template accepts a lot more directives than just `U`. It can be used to decipher all kinds of data. We can add a binary string to the mix with "A".

```
>> [65, 128514, 90, '!'].pack('UUUA')
=> "A\xF0\x9F\x98\x82Z!"
```

Wait, what happened there? We lost our emoji. Adding a binary string turned our into output into binary.

```
>> [65, 128514, 90, '!'].pack('UUUA').encoding
=> #<Encoding:ASCII-8BIT>
```

We know that what we have is Unicode-safe, so we can force the encoding into what we want.

```
>> [65, 128514, 90, '!'].pack('UUUA').force_encoding('UTF-8')
=> "A😂Z!"
```

The `pack` method is meant to create a binary sequence based on the contents of an array and a template. We're focusing on using it to put together UTF-8 strings. This'll be useful if we ever need to manipulate the code points of a string. We can do the work and rely on `pack` to put everything back together.

```
>> "A😂Z".codepoints.map { |cp| cp + 1 }.pack('U*')
=> "B😃["
```

Escaping Escaping

Escaping things sucks. Sometimes the \ chaos is simply too much to bear. If we're escaping quotes we could switch the type used to create the string, swapping double quotes for single quotes or visa versa. Of course, that'll also change how the string behaves. What if the string contains both types of quotes?

One common place I see this is strings that contain HTML and need interpolation. The results get ugly fast.

```
"<input type=\"text\" name=\"city\" value=\"#{value}\" />"
```

Ruby has an escape hatch for just such a situation. Instead of double quotes we can use `%Q` (or its alias `%`) to create our string. The `%Q` is immediately followed by a delimiter that we'll repeat to end the string. Any non-alphanumeric ASCII character can act as a delimiter.

```
%Q#Hello reader!#
```

You can even use a space! (Show below as "␣".)

```
>> %Q␣Hello\␣there!
.. ␣
=> "Hello there!\n"
```

If we use `(`, `[`, `{`, or `<` we'll have to use their closing companion instead of repeating the initial character. It creates a pleasant symmetry.

```
%Q(Hello reader!)
```

Let's look at that first example again with `%Q` instead of double quotes.

```
%Q(<input type="text" name="city" value="#{value}" />)
```

It's much easier to read this time.

We can still use \ to escape our delimiter. With paired delimiters we can even skip the \ as long as the pair is balanced inside the string.

```
%Q(Hello (or Bonjour) world!)
```

It's weird though. In cases like these we're better off changing out the delimiters.

We can do the same thing for single quoted strings with `%q`.

```
>> puts %q(a\nb)
a\nb
```

%w and %W

With `%q` and `%Q` we create single strings. If we change those to `%w` and `%W` we can create an array of strings. I know, technically we're creating arrays in a book about strings but they're pretty helpful. Each creates an array of strings where whitespace separates the strings.

```
>> %w[one two three]
=> ["one", "two", "three"]
```

I like to use `%w` when creating a list of strings because you can put each entry on its own line. This makes it easy to read and edit. I also prefer brackets over parenthesis for my default delimiter. They're unlikely to show up in the strings being created and they look more like arrays.

```
>> last_names = %w[
..    Picard
..    Riker
..    Troi
.. ]
=> ["Picard", "Riker", "Troi"]
```

If one of our names had a space, we're not prohibited from using `%w`. We can escape the space like we would with a single quote inside a raw string. By using `\` we can escape the space immediately after it.

```
>> last_names = %w[
..    La\ Forge
..    Picard
..    Riker
..    Troi
.. ]
=> ["La Forge", "Picard", "Riker", "Troi"]
```

The lower-case `%w` version creates raw strings. This means escape codes like `\n` are inserted as literal characters. If we want our strings to work like processed strings, we'll switch to `%W`. With that we'll be able to use escape codes and interpolation.

```
>> %W[
..    one\ntwo
..    E\sis\s#{Math::E}
.. ]
=> ["one\ntwo", "E is 2.718281828459045"]
```

Both of these can help remove the noise of quote after quote while creating a list of strings.

?F+?o+?r+?e+?!

What you see there is valid Ruby code.

Let's back up. In time. The year is 2008 and we're using Ruby 1.8.7. We're running Ruby Enterprise Edition because we're awesome and we know what's up.

We open IRB and snag the first character from a string.

64

```
>> 'Aaron'[0]
=> 65
```

That's the US-ASCII code point for "A" (remember, strings are US-ASCII in 1.8.7).
Let's say that we have a list of names. We want to find every name that starts with
a "D".

```
>> %w[Zack Miri Delaney Bubbles Deacon].select do |name|
..    name[0] == 'D'
.. end
=> []
```

Oh, right, in 1.8.7 the accessor returns a code point instead of a letter. We could
swap out `'D'` for `68` but most people haven't memorized the US-ASCII table and
won't be able to suss out what we're doing. We could call `'D'.ord`. It's readable
but a bit noisy.

Situations like this will come up fairly often and Ruby wants the developer to be
happy. So, the `?` was implemented. It returned the code point of the character
following it.

```
>> %w[Zack Miri Delaney Bubbles Deacon].select do |name|
..    name[0] == ?D
.. end
=> ["Delaney", "Deacon"]
```

There was an eventual realization that returning the code point wasn't all that use-
ful. In 1.9 the accessor was changed to return the character instead. To avoid break-
ing everything they changed `?` so that it also returned the character. Today, `?` is a
vestigial remnant that creates one character strings. It does still have one use: code
golf!

*

This is one of those features that comes in handy on occasion. Ruby will let us multiply a string. What I mean by that is it'll make n copies of the string and concatenate them.

```
>> 'na' * 16 + ' Batman!'
=> "nananananananananananananananana Batman!"
```

When I'm debugging code there are times I'll use a proper debugger. Other times I litter my code with `puts` statements to check a particular value or that a block has been reached. It can be hard to see my output when it's mixed in with a bunch of other input flying by.

I'll often box my input between two long, easy-to-distinguish lines of characters. It also makes it easier to search backwards in the output to find values if they fly by too quickly.

```
puts '#' * 20
puts some_variable
puts '#' * 20
```

The third line of the progress bar from the Escaping section made use of `*`.

```
print ('=' * (percent / 5)).ljust(20)
```

It prints between 0 and 20 equal signs depending on the completion percentage. Each equal sign represents five percent of the total progress.

It can also make code more readable. Quick, how many "#" are there?

```
separator = '#############################'
```

How many now?

```
separator = '#' * 30
```

Building Up Strings

Sometimes strings aren't constructed all at once. They're built up as we loop through data, or handed from one piece of code to the next. When it comes to adding content to a string, Ruby has no shortage of ways to do it. We can choose between `<<`, `concat`, `prepend`, `+`, and ' '. Yes, that last one is a space.

These methods can be grouped into mutating and non-mutating. With `<<`, `concat`, and `prepend`, new content is added to the existing string. The first two are aliases and add to the end of the string.

```
>> a = 'a'
=> "a"
>> a << 'b'
=> "ab"
>> a.concat('c')
=> "abc"
>> a
=> "abc"
```

We can do the opposite and add to the start of the string with `prepend`.

```
>> a = 'a'
=> "a"
>> a.prepend('b')
=> "ba"
>> a
=> "ba"
```

Mutating strings is generally something we'll only do to strings we instantiated. If we create a method that mutates an argument then we've added an unexpected side-effect. Let's say we're generating emails for a charity. Each email has a salutation, a body, and a closing. We'll focus on the salutation and closing.

```
def salutation(first_name)
```

```
    first_name.prepend('Dear ')
end

def closing(first_name)
  "Thank you #{first_name} for your generous donation."
end
```

What happens when we send one of our donors through these methods?

```
>> salutation(donor.first_name)
=> "Dear Aaron"
>> closing(donor.first_name)
=> "Thank you Dear Aaron for your generous donation."
```

Notice how our closing is thanking "Dear Aaron". Oops. In the `salutation` method, `prepend` directly altered the `first_name` string. When methods receive arguments they need to be careful about mutating them. I try to avoid it as much as possible.

We can avoid mutating a string by interpolating it into another string or by using +.

```
>> a = 'a'
=> "a"
>> a + 'b'
=> "ab"
>> a
=> "a"
```

It's tempting to avoid ever mutating strings. If we only ever create new ones then we never have to worry about seeing a "Dear Aaron"-type situation. The downside of this approach is performance.

Mutating a string uses less memory and runs faster. Most of what we do is a small number of changes to small strings. The performance gains will be negligible. With larger strings and especially when we add lots of content, it can make a difference.

When the Ruby interpreter allocates memory for a string it takes more than it needs.

It's an optimization that trades memory for speed. Bytes can be added to the string without having to allocate a new chunk of memory. If we add enough bytes Ruby is eventually forced to allocate a new bigger chunk of memory to hold our large string. Ruby takes care of this without ever telling us. Our string keeps growing and we don't have to think about it.

If we knew we were going to make a lot of changes it might be helpful to be able to tell Ruby that we'll need a bigger string from the start. Then Ruby can avoid repeatedly allocating new larger chunks of memory. Instead it would allocate one giant chunk at the beginning. We can do exactly that with `String.new`.

It takes a `:capacity` option which accepts a number of bytes to allocate. If we know the size ahead of time or can make a good guess, we'll avoid those memory allocations.

Let's compare doing lots of work without using mutation, while using mutation, and while using mutation with a set capacity.

```ruby
require 'benchmark/ips'

CYCLES = 1_000
chars = 'a' * 200

Benchmark.ips do |b|
  b.report('no mutation') do
    a = ''
    CYCLES.times { a += chars }
  end

  b.report('mutation') do
    a = ''
    CYCLES.times { a << chars }
  end

  b.report('mutation w/ capacity') do
```

```
    a = String.new(capacity: CYCLES * chars.length)
    CYCLES.times { a << chars }
  end

  b.compare!
end
```

Running this benchmark gave me:

```
mutation w/ capacity:  5728.3 i/s
           mutation:  4912.3 i/s - 1.17x  slower
        no mutation:    22.7 i/s - 252.78x  slower
```

We can see that avoiding mutation would really cost us. Without mutation our code was nearly 253 times slower than it needed to be. We can also see that while the gains are smaller, adding a capacity did provide a boost.

This benchmark is an example of how mutation and, to a lesser extent, capacity can affect execution speed. The numbers won't look this way for every bit of code. If you're looking to speed up code you should always do your own benchmark. Make sure the optimizations you're making work for the code you're writing.

Another interesting optimization is space concatenation. Ruby will concatenate a series of strings, each separated by zero or more spaces.

```
>> 'a''b' 'c'        'd'
=> "abcd"
```

These have to be string literals. It won't work with variables. It will work with single character ? strings or %q and %Q but only if they come first. Otherwise, Ruby attempts to parse the ? as the start of a ternary and %q as a modulus operation.

```
>> ?a 'b'
=> "ab"
>> 'a' ?b
```

70

```
SyntaxError: syntax error, unexpected end-of-input, expecting
    ':'

>> %q(a) 'b'
=> "ab"
>> 'a' %q(b)
NameError: undefined local variable or method `b' for main:
    Object
```

I referred to this as an optimization because it's faster than +. Instead of making each string a separate object, the Ruby interpreter reads through the strings and spaces until the end. At that point it makes the string object and returns it. When we use a + for two strings, the interpreter ends up creating 3 strings.

```
>> report = MemoryProfiler.report { 'a' + 'b' }.pretty_print
Total allocated: 120 bytes (3 objects)
```

Sam Saffron's MemoryProfiler gem provides an easy to use wrapper around Ruby's built in memory introspection. It reports that three objects have been created. The 'a' and 'b' were combined to make a third object 'ab'. Doing the same thing with a space instead we see that only the final 'ab' is created in memory.

```
>> report = MemoryProfiler.report { 'a' 'b' }.pretty_print
Total allocated: 40 bytes (1 objects)
```

Space concatenation was added to make it easy to create multi-line strings without having to pay the price of using + a bunch of times. Strings were broken across lines. They would end with a \ to tell the parser to treat them as one line.

```
>> 'a' \
.. 'b'
=> "ab"
```

If the strings are on one line then why separate them? If we want to format the string over multiple lines for better readability well... there are better ways.

Heredocs

If you don't recognize the name you'll certainly know one when you see it.

```
>> <<-TXT
.. For use: one off script, never tested.
.. TXT
=> "For use: one off script, never tested.\n"
```

The heredoc (a.k.a "here document") was originally introduced in the Bourne shell back in 1977. It was a way to insert a document right into the middle of a script. Instead of having to keep the doc elsewhere it was, well... right here. Other Unix shells adopted the concept and it eventually made its way into languages like Perl and Ruby.

In Ruby, all heredocs start with <<. The most minimal heredoc immediately follows that with an identifier, some content on the lines that follow, and the identifier again at the start of the line.

```
<<ID
Here's some content.
ID
```

Any lines between the two identifiers, ID, are collapsed and put into a string. The string is returned at the location of the << and the opening identifier. This means we can use multiple heredocs on a single line and they'll collapse in order.

```
>> [<<FIRST, <<SECOND]
.. This is first.
.. FIRST
.. This is second
.. SECOND
=> ["This is first.\n", "This is second\n"]
```

Ruby will read everything from the start of the line following the identifier until it

reaches a line that starts with the identifier. When I say, "the start of a line" I mean the literal first character. Spaces count.

Breaking out a series of heredocs could be useful for separating sections of a long document or subqueries in a SQL string.

```
>> "SELECT * FROM (#{<<QUERY_1}) JOIN (#{<<QUERY_2})".delete("
   \n")
.. SELECT * FROM A
.. QUERY_1
.. SELECT * FROM B
.. QUERY_2
=> "SELECT * FROM (SELECT * FROM A) JOIN (SELECT * FROM B)"
```

In these examples, the closing identifier has to be at the start of the line. That's not the way we write code. Code is indented. I mean, we're not monsters.

Heredocs have two option modifiers that can be used between the << and the opening identifier. The first is the – which lets us indent our closing identifier.

```
>> <<-WHITESPACE
.. This is some text.
..                WHITESPACE
=> "This is some text.\n"
```

Sadly any whitespace in our content is retained. We can safely indent the identifier but not the content.

```
>> <<-WHITESPACE
..        Ahhhh, too much whitespace!
..    WHITESPACE
=> "        Ahhhh, too much whitespace!\n"
```

In Ruby 2.3 we got the ~ modifier. This one will let us indent both the identifier *and* the content. It will find the line with the lease amount of whitespace preceding it and strip that much space from every line of content.

```
>> puts <<~WHITESPACE
..            first
..              second
..                third
..        WHITESPACE
first
  second
    third
=> nil
```

All the space was stripped from the start of the line with "first". The other two lines then had that same amount of space removed. Before ~ came along, people were left with two options. In Rails land, `strip_heredoc` performed the same operation as the ~ in 2.3.

```
>> puts <<-WHITESPACE.strip_heredoc
..            first
..              second
..                third
..        WHITESPACE
first
  second
    third
=> nil
```

Your other option was to hide under your desk, hold your knees, and sob uncontrollably knowing that your powerless to stop these spaces from ruining otherwise beautiful code. No? Just me? Moving on.

You may have noticed that I've used uppercase or shouting case or whatever you want to call it for my identifiers. That's not a requirement. It is a common practice. By doing that you're not likely to accidentally start a line with the identifier. It also makes them stand out in the text.

Heredoc identifiers should be like variable names. They should help to identify the

content inside. Coming across `TERMS_AND_CONDITIONS` is much more descriptive than `TXT`. The exception to this is when your code contains other code. It can be helpful to mark that content by its type. Marking a heredoc with labels like `SQL`, `HTML`, and `JAVASCRIPT` hints that foreign code is in this Ruby file. Some editors will read those labels and attempt to properly highlight the content of the heredoc.

By default, heredocs are treated like a double quoted string. We can use escape codes and interpolate right inside the content. By putting single quotes around the identifier we can make it behave like a single quoted string.

```
>> NOT = 'interpolated'
=> "interpolated"

>> <<~CONTENT
..    this is #{NOT}
.. CONTENT
=> "this is interpolated\n"

>> <<~'CONTENT'
..    this is #{NOT}
.. CONTENT
=> "this is \#{NOT}\n"
```

You can also use backticks if you want to execute the contents as a multi-line shell command.

```
<<~`COMMAND`
  apt-get update -qq \
    && apt-get install -y \
      build-essential \
      cron
    && apt-get clean
COMMAND
```

It's possible to use explicit double quotes as well. Given that it's the default behavior, it's not something you're likely to come across. Finally, remember that Ruby

supports Unicode characters and they can be identifiers.

```
>> <<~💩
..    It'll be a cold day in hell before I actually do this.
.. 💩
=> "It'll be a cold day in hell before I actually do this.\n"
```

You should do this in shared code. People will love it.

Recap

There are so many ways to create a string. Each has it's own use. Raw strings give you exactly what you type. Processed strings are dynamic and have their own language of escape codes and interpolation options. We can add to or create strings with code points, question marks, and asterisks. When we have giant blocks of text, we know all of the tricks of the heredoc.

There are times when we won't be creating new strings. We'll need to turn other data into a string. Next, we'll look at converting other objects into strings and the differences between implicit and explicit conversion.

9 | Conversion

We've covered a million ways to create a string but what if we want to convert something else into a string? To start, we can pass whatever it is to `Kernel#String`.

```
>> String(1)
=> "1"
```

When we call `String` it'll first try calling `to_str` and if that doesn't work it'll call `to_s`. Great, but what do those do?

to_s

Nearly every object has a `to_s` method that stringifies it. It's the method that gets called during string interpolation.

```
>> 1.to_s
=> "1"
>> 1.1.to_s
=> "1.1"
>> :a.to_s
=> "a"
>> [1, 2].to_s
=> "[1, 2]"
```

For most classes it's a fairly straight forward conversion. There are a couple of classes that take it further and let us provide arguments to `to_s`.

Integers accept a base that you would like the number converted into.

```
>> 35.to_s(2)
=> "100011"
>> 35.to_s(16)
=> "23"
>> 35.to_s(10)
=> "35"
```

The default base is `10` but we can pick anything between `2` and `36`. The other direction also works in case you want to turn a string into an integer.

```
>> '100011'.to_i(2)
=> 35
```

Another interesting one is `BigDecimal`.

```
>> BigDecimal.new('3.141592').to_s
=> "0.3141592E1"
```

Unless you've done this before, that's probably not what you expected to see. The good news is we can pass a format for our output. The default is `E` which gives the scientific notation and forces a leading zero. Using `F` produces a float-like notation which is likely closer to what you expected.

```
>> BigDecimal.new('3.141592').to_s('F')
=> "3.141592"
```

The format can have an optional leading + or space. Whichever you pick gets added to the front of positive numbers. Negative numbers will always have a leading –.

```
>> BigDecimal.new('3.141592').to_s('+F')
=> "+3.141592"
>> BigDecimal.new('3.141592').to_s(' F')
```

```
=> " 3.141592"
>> BigDecimal.new('-3.141592').to_s(' F')
=> "-3.141592"
```

We can also give it a number and it'll group the digits by that amount. The digits to the left of the decimal point are grouped independently of those to the right. Both are grouped from the left which can lead to some interesting outputs.

```
>> BigDecimal.new('10000.00001').to_s(3)
=> "0.100 000 000 1E5"
>> BigDecimal.new('10000.00001').to_s('3F')
=> "100 00.000 01"
>> BigDecimal.new('10000.00001').to_s('+3F')
=> "+100 00.000 01"
```

Excluding the rare use of `BasicObject`, every class we create will have a `to_s`. If there's some benefit to extending that functionality then it's worth considering. Certainly we can see there's precedent to do so.

to_str

Every object may have `to_s` but almost none of them have `to_str`. Nor should they. With `to_str` we're asking the object if it can be implicitly converted into a string. That means the object should be able to stand-in for a `String`.

Earlier we saw a number converted to a string using `to_s`. Numbers can be shown as a string but they aren't the same as strings. That's why they don't implement `to_str`.

```
>> 1.to_s
=> "1"
>> 1.to_str
NoMethodError: undefined method `to_str' for 1:Fixnum
```

Let's build a class called `CodePointList`. It'll take a series of Unicode code points that represent characters in a particular order. If that sounds a lot like a string that's because it is.

```ruby
class CodePointList
  def initialize(*code_points)
    @code_points = code_points
  end

  def to_s
    @code_points.pack('U*')
  end
end
```

For all intents and purposes, our `CodePointList` creates strings. Its focus is on code points instead of characters but the result is still a string.

```
>> "hello #{CodePointList.new(121, 111, 117)}"
=> "hello you"
```

If we try to use it like a `String` we're going to run into problems. To see what I mean, let's try concatenating an instance of `CodePointList` using +.

```
>> "hello " + CodePointList.new(121, 111, 117)
TypeError: no implicit conversion of CodePointList into String
```

The + is a method like any other. It takes an argument and attempts to append it to the instance it's called on. The code above could be written as:

```
"hello ".+(CodePointList.new(121, 111, 117))
```

The + method is expecting to be able to convert the object given into a string. The error isn't that the object isn't a `String`. It's that there is no "implicit conversion of CodePointList into String".

We can fix this by adding one line into the `CodePointList` class.

```
alias_method :to_str, :to_s
```

Now the class has a `to_str` method and Ruby will know that it should be usable as a string.

```
>> "hello " + CodePointList.new(121, 111, 117)
=> "hello you"
```

If we really want `CodePointList` to work like a string we'll have to mimic existing methods. Right now they'll throw errors.

```
>> CodePointList.new(121, 111, 117) + ' are great!'
NoMethodError: undefined method `+' for #<CodePointList:0
    x007f9b7d99d048 @code_points=[121, 111, 117]>
```

To implement our own + method we'll also use `to_str`.

```
def +(other)
  new_code_points = @code_points + other.to_str.codepoints
  self.class.new(*new_code_points)
end
```

By using `to_str` we've insured that this works with `CodePointList`, `String`, and any implicit string class that someone creates. Now we can use + to add other strings to our string.

```
>> sentence = CodePointList.new(121, 111, 117) + ' are great!'
=> #<CodePointList:0x007fa7aa3ea6f8 @code_points=[121, 111,
    117, 32, 97, 114, 101, 32, 103, 114, 101, 97, 116, 33]>
>> sentence.to_s
=> "you are great!"
```

If we try to add a non-string object it'll throw a `NoMethodError` because `other` won't have implemented `to_str`. We could rescue that error and throw our own. We could also use `String.try_convert` which tries `to_str` and if it's unavailable returns `nil`. From there we can raise our own error.

```
def +(other)
  other_str = String.try_convert(other)

  unless other_str
    raise TypeError, "no implicit conversion of #{other.class}
        into String"
  end

  new_code_points = @code_points + other_str.codepoints
  self.class.new(*new_code_points)
end
```

With this we get the exact behavior of `String#+`.

```
>> CodePointList.new(121, 111, 117) + ' are great!'
=> #<CodePointList:0x007fa7aa3ea6f8 @code_points=[121, 111,
   117, 32, 97, 114, 101, 32, 103, 114, 101, 97, 116, 33]>
>> CodePointList.new(121, 111, 117) + nil
TypeError: no implicit conversion of NilClass into String
```

The pattern we used for + could be extracted with meta-programming to easily implement lots of the existing `String` methods.

Recap

By exploring `Strings`, `to_s`, `to_str`, and `String.try_convert` we've covered all the avenues of string conversion. With `to_str` we saw that implicit conversion is useful for string-like objects. We even made our own fancy `CodePointList` string-like object that Ruby will treat like a string. Explicit conversion via `to_s` is omnipresent and in some cases provides formatting options.

We've covered creating strings from nothing and converting other objects into strings. We need to address one more item when producing a new string: mutability.

10 | Freezing

Is it me or is it freezing in here? - my wife

Strings created in Ruby are almost always mutable but that may be changing. Mutable means that the contents of the string can change. If they were immutable we'd have to create a new string when we wanted to change something. We can make objects immutable by freezing them. Frozen things don't move. They don't change.

At first this might seem like a disadvantage. To change a frozen string we have to create a new one. It seems wasteful. Of course if we have multiple immutable strings with the same contents, the interpreter can reuse one object for all of them. If we have lots of strings that don't change, making them immutable and reusing them will save memory. If we have one string that we change a bunch it'll be faster to change the original rather than constantly having to make new copies for each update. Each approach has its own benefit.

Immutable strings have one other killer feature: they can't be changed. I know, I know, that's obvious but it's also powerful. With mutable strings we have to trust the code that interacts with them. The problem is we might not be able to. Passing around a mutable string is like handing a rare mint condition playing card to a room full of kindergarteners. It won't be mint when you get it back. Take that same card, seal it in an indestructible transparent case, and then pass it around. You can be confident that it'll come back undamaged. That's the advantage of immutability.

Instead of creating a web of trust within our code and hoping that it holds, we can make the string immutable and not worry about it. It also means we can share that string between threads without any issues. We won't have one thread changing the string half way through the other one using it.

To make a string immutable we call `freeze` on it. We can also check to see if it's immutable with `frozen?`.

```
>> quote = 'Let it go.'
=> "Let it go."
>> quote.freeze
=> "Let it go."
>> quote.frozen?
=> true
```

Once we've done this, the change is permanent. There's no way to thaw it back out. Which make sense if you think about it. The point of freezing a string is to protect it. Undoing that would destroy the guarantee of safety.

If you need to mutate a frozen object, the best you can do is make a new non-frozen copy. You can do this with `dup` or `clone(freeze: false)`. Both return a new copy that you can alter without risking damage to the original. Used this way, the only difference between the two is that `clone` will retain any singleton methods added to the original object.

It's worth noting that this is true for strings but not for any object that reference others. For example, the values in arrays and hashes can be altered even if the array or hash is frozen.

```
>> a = %w[first second].freeze
=> ["first", "second"]
>> a.frozen?
=> true
>> a[0].replace('one')
=> "one"
```

```
>> a[1].replace('two')
=> "two"
>> a
=> ["one", "two"]
>> a.push('three')
RuntimeError: can't modify frozen Array
```

By default strings aren't frozen but maybe they should be. It turns out there are a lot of repeated strings within our code. How many strings are created when we run the following code?

```
>> [%w[John Doe], %w[Jane Smith]].each do |names|
..    puts names.join(' ')
.. end
John Doe
Jane Smith
=> [["John", "Doe"], ["Jane", "Smith"]]
```

We'll use the MemoryProfiler gem we discussed in the Creation chapter to check.

```
allocated objects by class
-------------------------------
        8  String
        3  Array
```

We're creating eight strings in this code. Let's count them up. Each name is a string so that gives us four. The `' '` that we used in `join` bumps the total to five. Then we have two output strings that are the result of the `join`. That's only seven. We're missing one.

The missing string is the `' '` passed to `join`. We counted it one time but it's actually created twice. Every time through the loop creates a new instance of the `' '` string. If we have 100 name pairs to join, the interpreter will create 100 different `' '` strings. By freezing the string, Ruby knows that it might get the chance to reuse it and keeps the string around. Instead of 2 strings, or 100 strings, it'll only ever

make the one instance.

```
>> [['John', 'Doe'], ['Jane', 'Smith']].each do |names|
..   puts names.join(' '.freeze)
.. end
John Doe
Jane Smith
=> [["John", "Doe"], ["Jane", "Smith"]]
```

How many other places in the code might use ' '? Over a large number of objects this can really start to add up. If Ruby created immutable strings by default, we'd get this benefit without ever having to think about it.

Performance Gains

After it was pointed out to the community, gem authors began leveraging frozen strings to optimize their code. Largely led by Richard Schneeman, Rails underwent an overhaul to freeze every string they could. In one commit[1], he froze a series of strings that shaved about a half a millisecond off of each Rails request. It may not sound like much but that's about 1 second for every 220 requests and the only change was freezing some strings.

This spate of freezing prompted Matz to consider a change. Ruby 2.3 included an experiment that allows developers to make strings immutable by default. If it turns out to be useful, this change may find its way into Ruby as the new default. It's a big change and Matz has made it clear that if it happens, and that's a big if, it'll be in a major revision of Ruby. That means the earliest it could happen would be Ruby 3.0.

---

[1]https://github.com/rails/rails/commit/5bb1d4d288d019e276335465d0389fd2f5246bfd

Flipping the Switch

There are a couple of ways to turn on this experimental feature. We can call `ruby` with the `--enable-frozen-string-literal` option. At that point all strings created using quotes (or their `%` equivalents) will be frozen. This approach comes with two problems. First, we might not be in control of how `ruby` is called. If we're writing a gem we don't get to dictate how people run our code. Second, it's a pretty all or nothing move. What if we have gems that aren't ready to have their strings frozen?

We can also turn on the frozen string experiment with a pragma at the top of any file. By adding `# frozen_string_literal: true` to the top of the file we turn it on but only for the code in that file.

```
# frozen_string_literal: true


'This string is frozen!'
```

When I want to learn about a new feature I like to try it out in IRB. Unfortunately, we can't pass `--enable-frozen-string-literal` directly to IRB. Instead, we'll have to pass it through the `RUBYOPT` environment variable.

```
$ RUBYOPT=--enable-frozen-string-literal irb
>> 'This string is frozen!'.frozen?
=> true
```

You Still Have Choices

When we enable the experiment we're only changing the default for new strings. We still have mutable strings. No matter what we do, `String.new` will always produce a mutable string.

```
$ RUBYOPT=--enable-frozen-string-literal irb
>> ''.frozen?
=> true
>> String.new('').frozen?
=> false
```

In addition to the experiment, Ruby 2.3 also introduced two new unary methods on
`String`. Preceding a string with – will cause it to be frozen. The same thing can be
done with + to create a non-frozen string. Once again, these work no matter what
the default is.

```
>> (-'').frozen?
=> true
>> (+'').frozen?
=> false
```

The parenthesis are necessary if you want to chain methods onto the string. Other-
wise Ruby will parse the `''.frozen?` and then try to call the + or – unary operator.

```
>> +''.frozen?
NoMethodError: undefined method `+@' for true:TrueClass
```

These + and – operators can also be applied to heredoc strings.

```
>> (-<<-TXT).frozen?
This is frozen!
TXT
=> true
```

In fact, you can apply them to any string you have.

```
>> str = ''
=> ""
>> str.frozen?
=> false
>> (-str).frozen?
=> true
```

```
>> str.frozen?
=> false
```

It might seem like – is nothing more than an alias for `freeze`. The two are slightly different though. Rather than freezing the existing string, – creates a duplicate and then freezes the duplicate. That's why the last `str.frozen?` returns `false` instead of `true`.

What About Other Methods?

What happens to the other `String` methods when you turn this on? Nothing. Even the methods that create new strings will return them as mutable strings.

```
$ RUBYOPT=--enable-frozen-string-literal irb
>> ['a', 'b'].join(' ').frozen?
=> false
```

Unfortunately, this makes it easy to create mutable strings when you might not want to. The good news is these strings are less likely to be pre-existing so we'll still get the majority of the memory savings. We only really lose the safety. Of course, we can always force that.

```
>> -['a', 'b'].join(' ')
=> "a b"
>> _.frozen?
=> true
```

Recap

We've exhausted the topic of string creation. Now we need to know what to do when someone else gives us a string. We'll need to know how to adjust it for our own needs. Before even that, we need to know how to examine what we've been given.

11 | Examination

Grab your magnifying glass and tape measure because we're going in. We've created strings and converted other objects into strings. Now that we have strings, we need to be able to look inside.

What Character Is That?

There are times where everything will look fine but your code won't work. This can happen because characters are not always what they appear. Some characters are visually identical and others may be invalid. Code points can help you to peak under the hood and reveal what's really going on. Here we have a space that looks normal but is actually a hair width space. Typically it would display as a thinner than normal space but we're using a monospace font so there's no visible difference.

```
>> a = 'a' << 8202 << 'b'
=> "a b"
>> a.codepoints
=> [97, 8202, 98]
```

For short strings this works fine but with longer strings it can be hard to match the code points to the characters. In those cases, I recommend giving `dump` a try. It escapes special characters and, most importantly, replaces all non-printing and

non-US-ASCII characters.

```
>> "One space is hair width.".dump
=> "\"One space is\\u200Ahair width.\""
```

We couldn't see it but, with `dump` we were able to easily spot the problem.

Finding Characters

There are two methods that will search for a substring. A substring is a string whose entirety appears within the contents of another string. Both methods are effective but they differ in what they return.

```
>> 'Book'.include?('o')
=> true
>> 'Book'.include?('ok')
=> true
>> 'Book'.include?('okk')
=> false

>> 'Book'['o']
=> "o"
>> 'Book'['ok']
=> "ok"
>> 'Book'['okk']
=> nil
```

As the question mark indicates, `include?` returns a boolean value. The accessor method (i.e. `[]`) returns the match or `nil` if there isn't one. Both of these will return truthy values when given an empty string. An empty string is always a substring.

With `start_with?` and `end_with?` we can do the same thing as `include?` but we can limit the check to the start or end of the string.

```
>> 'book'.start_with?('b')
=> true
>> 'book'.end_with?('ok')
=> true
```

Anything beyond this and we'll need to reach for a regular expression. With regular expressions we can check the start or end of the string. We can check word boundaries to guarantee we find the exact word we want. The possibilities are too numerous to list.

If we're only checking to see if a pattern exists, our best bet is `match?`. It takes a regular expression and an optional starting position. The optional position moves where the matching begins. Let's see if we can find two vowels in a row.

```
>> 'book'.match?(/[aeiou]{2}/)
=> true
>> 'book'.match?(/[aeiou]{2}/, 2)
=> false
```

This method was added in 2.4. In prior Ruby versions, the best alternative would have been `=~`. This is sometimes affectionately referred to as the "bacon cannon". It returns the position where the match starts. If no match is found it returns `nil`.

```
>> 'book' =~ /[aeiou]{2}/
=> 1
>> 'book' =~ /a/
=> nil
```

The `match?` method is faster than `=~` because it does not set any of the global variables associated with regular expressions. Most of the time you won't use those anyway.

Counting Characters

We can figure out the total number of characters in a string with either `size` or `length`. Remember that some characters might display as a single character but count separately in the size of the string.

```
>> 'é'.unicode_normalize(:nfd).size
=> 2
```

I'm using an NFD string but even NFC strings can display characters which require more than one string character to construct. At times we may want to know how many times a particular character appears. For that we can use `count`.

```
>> 'It was a pleasure to burn.'.count('a')
=> 3
```

Your intuition might be that `count` looks for substrings but that's not how it works.

```
>> 'It was a pleasure to burn.'.count('ea')
=> 5
```

There's one instance of "ea" in that string but there certainly aren't five. The `count` method takes what I like to call a "character set expression". The string is treated as though it was inside a regular expression and surrounded by brackets. You have to imagine it as `/[ea]/`. When you realize that it's counting every "e" and every "a" suddenly the 5 makes more sense.

We could check for all vowels by listing them out. Don't forget about the capital letters.

```
>> 'It was a pleasure to burn.'.count('aeiouAEIOU')
=> 9
```

With ^ we can negate the expression and find the number of non-vowel characters.

```
>> 'It was a pleasure to burn.'.count('^aeiouAEIOU')
=> 17
```

Character set expressions can also handle ranges of characters. We can find out how many letters there are by using –.

```
>> 'It was a pleasure to burn.'.count('a-zA-Z')
=> 20
```

We can see that 20 of the 26 characters are letters. We have the option of passing more than one character set expression to `count`. For a character to be counted it must pass every expression.

To count the number of constants we can combine two of the checks above. We'll look for every letter and then check to make sure it is not a vowel.

```
>> 'It was a pleasure to burn.'.count('a-zA-Z', '^aeiouAEIOU')
=> 11
```

This is our first foray into character set expressions but we'll see them again later.

Extracting Characters

We know it's in there but how do we get it out? When it comes to extracting characters, `[]` will take us a long way.

To start, we can give it an index and retrieve the character at that index.

```
>> 'Aaron'[0]
=> "A"
```

We can also start from the end of the string by using negative indexes.

```
>> 'Aaron'[-1]
```

```
=> "n"
```

A complete breakdown of the available indexes for "Aaron" looks like this:

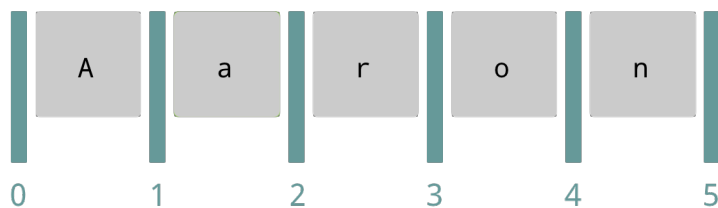| A	a	r	o	n
0	1	2	3	4
-5	-4	-3	-2	-1

Anything beyond `4` or `-5` will return `nil`.

```
>> 'Aaron'[5]
=> nil
>> 'Aaron'[-6]
=> nil
```

Maybe we want to grab a few characters. There are two ways to do it. The first is to provide a starting position and the number of characters we want.

```
>> 'Aaron'[2, 3]
=> "ron"
```

Did you notice that I switched from talking about an index to talking about a position? When I use the word index, I'm referring to the location of a single character in the string. A position is a location between two characters or between a character and the start or end of the string. Here are the positions for "Aaron":

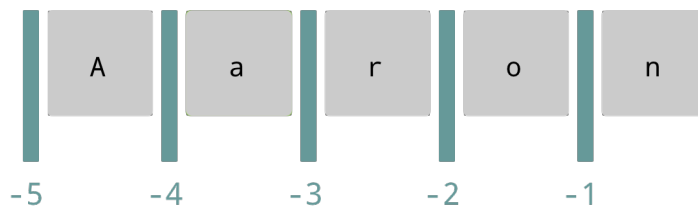|	A	a	r	o	n	
0		1	2	3	4	5

We can test this out.

```
>> 'Aaron'[5]
=> nil
>> 'Aaron'[5, 1]
=> ""
```

The index 5 is beyond the end of the string. The position 5 exists between the "n" and the end of the string. The string content between those two is empty. If we step one position further we no longer get an empty string.

```
>> 'Aaron'[6, 1]
=> nil
```

If you don't know about this it can be surprising and difficult to debug.

Like indexes, negative positions start from the end of the string.

The number of characters to extract always remains positive. We can only extract characters to the right of a position.

```
>> 'Aaron'[-3, 3]
=> "ron"
```

Asking for more characters than there are doesn't hurt anything. It'll give us as many characters as it can.

```
>> 'Aaron'[-3, 10]
=> "ron"
```

There's a second way to get a range of characters. I'll tell you now that I prefer what we just did to what's coming. You've been warned.

Ranges use start and end positions to cut out a section of the string.

```
>> 'Aaron'[1..4]
=> "aron"
```

An inclusive range (i.e. . .) extracts each character starting with the character after position 1 and ending with the character **after** position 4. Here are the positions again:
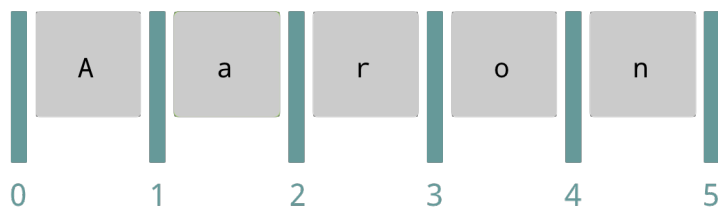
Like before, we can extract the characters between position 5 and the end of the string.

```
>> 'Aaron'[5..5]
=> ""
```

If we want the range to only include characters that fall between two positions then we need an exclusive range (i.e. . . .).

```
>> 'Aaron'[1...4]
=> "aro"
```

Also like before, we can use negative positions. The negative positions are just a way to reference the position from the other direction. It doesn't matter if we use a positive or negative position, the starting position must come before the ending position.

```
>> 'Aaron'[1...4]
=> "aro"
>> 'Aaron'[1...-1]
=> "aro"
```

```
>> 'Aaron'[-4...-1]
=> "aro"
>> 'Aaron'[-4...4]
=> "aro"
```

If we pick a starting position within the string that comes after the ending position, we'll always get an empty string.

```
>> 'Aaron'[2..1]
=> ""
```

Positions are a bit confusing but it works this way for a reason. Both of these approaches have equivalent assignment operations. Using them we can insert or replace characters. For assignment, referencing the space between characters is a must. It's the only way to accurately insert a character.

All of this is great but if we need more flexibility we'll turn again to regular expressions. The accessor can also handle that! When given a regular expression, it'll return the match or `nil`.

```
>> 'Aaron'[/[aeiou]+/i]
=> "Aa"
```

However, when given a capture index or name it will return that value.

```
>> 'Aaron'[/\A(.).*(.)\z/i, 1]
=> "A"
>> 'Aaron'[/\A(.).*(.)\z/i, 2]
=> "n"

>> 'Aaron'[/\A(?<first>.).*(?<last>.)\z/i, :first]
=> "A"
>> 'Aaron'[/\A(?<first>.).*(?<last>.)\z/i, :last]
=> "n"
```

Passing `0` will provide you the entire match as if you passed nothing. After that

you start accessing the capture groups. Passing a capture index that doesn't exist
returns `nil` but a named capture that doesn't exist throws an error.

```
>> 'Aaron'[/\A(.).*(.).\z/i, 10]
=> nil
>> 'Aaron'[/\A(?<first>.).*(?<last>.)\z/i, :ten]
IndexError: undefined group name reference: ten
```

I like capture groups because they provide a descriptive name and you can't acci-
dentally get the wrong one.

If we need more than one capture we'll have to switch over to `match`. It returns a
`MatchData` instance that provides all sorts of nifty information.

```
>> m = 'Aaron'.match(/\A(?<first>.).*(?<last>.)\z/)
=> #<MatchData "Aaron" first:"A" last:"n">
```

There are a lot of ways to access the captures. We used named captures in our reg-
ular expression but the captures can still be referenced by their index.

```
>> m[:first]
=> "A"
>> m['first']
=> "A"
>> m[1]
=> "A"
```

We can get multiple captures with the accessor or with `values_at`. The accessor
method accepts a starting index and a length or a range. By this point you shouldn't
be surprised to see it takes negatives.

```
>> m[1,2]
=> ["A", "n"]
>> m[-2,2]
=> ["A", "n"]
>> m[1..2]
=> ["A", "n"]
```

```
>> m[1..-1]
=> ["A", "n"]
```

Those don't allow us to use named captures and they also only select a contiguous set of values. With `values_at` we can pick and choose the captures we want in any order. We can mix named captures with indexes and even repeat captures if we want.

```
>> m.values_at(2, :first, 1, 2)
=> ["n", "A", "A", "n"]
```

If we want every capture we can use `captures` or `named_captures` to get all of them.

```
>> m.captures
=> ["A", "n"]
>> m.named_captures
=> {"first"=>"A", "last"=>"n"}
```

There's one case that `match` isn't suitable for. Extracting a series of values from a string based on a single pattern. For that we'll have to switch to `scan`.

```
>> '123 456 789'.scan(/\d+/)
=> ["123", "456", "789"]
```

It takes the provided pattern, finds a match, and then continues scanning for additional matches. We can also pass a block to make it act like `each`. We'll gain access to the matches and get back the original string instead of an array of matches.

```
>> '123 456 789'.scan(/\d+/) { |m| puts m.reverse }
321
654
987
=> "123 456 789"
```

If we're scanning very rigid data we could also use `scanf`. It takes a template string

like `sprintf` and returns an array containing each part of the template. It's only available after calling `require 'scanf'`.

```
>> require 'scanf'
=> true
>> '2017-01-01'.scanf('%4d-%2d-%2d')
=> [2017, 1, 1]
```

Unlike `scan`, it will not work around other characters and it'll only match one time.

```
>> 'Today is 2017-01-01'.scanf('%4d-%2d-%2d')
=> []
```

We can get multiple matches but only if we pass a block. The block is there to allow you to modify the output but it also triggers the `scanf` to repeat. This means we can get multiple matches by passing `{ |m| m }` as the block.

```
>> '2017-01-01 2017-01-02'.scanf('%4d-%2d-%2d') { |m| m }
=> [[2017, 1, 1], [2017, 1, 2]]
```

We could also pass the identity method, `itself`, and get the same effect.

```
>> '2017-01-01 2017-01-02'.scanf('%4d-%2d-%2d', &:itself)
=> [[2017, 1, 1], [2017, 1, 2]]
```

In order for it to repeat the matches can only be separated by space characters. Any other characters will halt the matching.

As I mentioned before we could also alter the output.

```
>> '2017-01-01 2017-01-02'.scanf('%4d-%2d-%2d') do |year,
     month, day|
..    [day, month, year].join('/')
.. end
=> ["1/1/2017", "2/1/2017"]
```

In this example, we'll flip all of the bits in a binary string. We can easily do this with `scanf` and a block.

```
>> '10010'.scanf('%1d') { |d,| (d + 1) % 2 }.join
=> "01101"
```

With `scanf` we always get an array of values. This is the case even if there's only one match. The `|d,|` acts as a quick and dirty destructuring of that array. It gives us the first value. The rest of the block flips `0` to `1` and visa versa.

Sorting

Ruby sorts by walking both strings and checking each *byte*. Encodings don't matter. Multi-byte characters don't matter. It's all about the byte.

Before 2.0 when Ruby still used ASCII as the default encoding, this resulted in a reasonably sane sort. Characters were sorted in a manor lovingly referred to as "asciibetical". It's a "good enough" method that quickly crumbles when faced with adversity.

Let's sort some numbers.
```
>> ('1'..'10').sort
=> ["1", "10", "2", "3", "4", "5", "6", "7", "8", "9"]
```

Ok so `"10"` comes before `"2"`. Maybe that's not so bad. If we wanted numeric sorting maybe we should have used numbers. Then again do we want to train our users to expect "10 Things I Hate About You" to come before "2 Fast 2 Furious"?

If you expected the problems to stop at numbers well, sorting letters is also a bit dicey. Uppercase letters always come before lowercase letters. That means "Zelda" will show up before "aardvark". Sorting without taking case into consideration is common enough that Ruby provides `casecmp` for doing exactly that.
```
>> ['aardvark', 'Zelda'].sort
```

```
=> ["Zelda", "aardvark"]
>> ['aardvark', 'Zelda'].sort(&:casecmp)
=> ["aardvark", "Zelda"]
```

Symbols and punctuation are scattered all over the ASCII table so good luck remembering how they sort.

```
>> ['A', 'a', '^', '~', '!'].sort
=> ["!", "A", "^", "a", "~"] # Wat?
```

When comparing bytes, if Ruby runs out of bytes the missing byte is considered smaller. This is why strings that start the same but are different lengths end up sorted by length.

```
>> ['abc', 'ab'].sort
=> ["ab", "abc"]
```

When sorting, Ruby uses <=> to compare the strings. Often called the "spaceship" operator because of its appearance, it returns one of four values. A -1 indicates that the first item is less than the second and order can be kept. With 0 we find out that the items are equal so, the order doesn't matter. A 1 shows that the first item is greater than the second and the order needs to be changed. Finally, a nil can be returned if the two items are not comparable.

```
>> 'a' <=> 'b'
=> -1
>> 'a' <=> 'a'
=> 0
>> 'b' <=> 'a'
=> 1
>> 'a' <=> 1
=> nil
```

It's import to remember that these aren't character comparisons. They're being compared by their bytes. The truth is we don't have a much better way. Numbers

have a natural order to them and we all learned our ABCs in school. That doesn't cover punctuation though. At least not anymore. What we now call the ampersand (i.e. &) used to be at the end of the alphabet. Children would sing "x, y, z, and per se and". Eventually "and per se and" started to slur into "ampersand". The name stuck even after we stopped ending our alphabets with it.

Even so, ampersand hardly represents the whole of our punctuation. Imagine how the song might go with everything thrown in.

"...w, x, y, and z, then comes period, exclamation, question mark, octothorp, colon, semi-colon, ellipses, ... (5 minutes later), and forward slash"

That's not even considering language issues. What order to accented characters go in? Does "語" come before or after "a"?

We can pass a block to `sort` and try to fix these issues but it gets complicated fast. Let's take a list of movies and start with plain old `sort`.

```
>> movies = [
..    "Amélie",
..    "10 Things I Hate About You",
..    "2 Fast 2 Furious",
..    "American Beauty"
.. ]
>> movies.sort
=> [
      "10 Things I Hate About You",
      "2 Fast 2 Furious",
      "American Beauty",
      "Amélie"
    ]
```

Most people would probably sort a "2" before a "10". They'd probably also ignore the accent and sort "Amélie" before "American Beauty". Here's `sort` with a block that fixes these issues.

```
>> movies.sort do |a, b|
..    a_words = a.split(/\s+/)
..    b_words = b.split(/\s+/)
..
..    numbers = /\A\d+\z/
..    normalize = ->(word) do
..      word.unicode_normalize(:nfkd).delete("\u0300-\u036f")
..    end
..
..    [a_words.size, b_words.size].max.times do |i|
..      a_word = a_words[i].to_s
..      b_word = b_words[i].to_s
..
..      comparison =
..        if a_word.match?(numbers) && b_word.match?(numbers)
..          a_word.to_i <=> b_word.to_i
..        else
..          normalize.call(a_word) <=> normalize.call(b_word)
..        end
..
..      break comparison if comparison != 0
..    end || 0
.. end
=> [
     "2 Fast 2 Furious",
     "10 Things I Hate About You",
     "Amélie",
     "American Beauty"
   ]
```

We have to compare each word. If both words are numbers then we compare them as numbers. If one or both are not numbers then we normalize them and strip accents. This will do a decent job but it's not even covering a bunch of cases.

It's unfortunate but the reality is we'll mostly sort the easy way and avoid this. Even if we design the perfect sorting method in Ruby, our databases will do their own

thing. If this is important then the best middle ground is to store a version where the letters are at least normalized.

Recap

We've discussed how to check a string for a character or substring. We can extract a portion of the string using either indexes or positions. Before moving on, make sure you're comfortable with character set expressions. We're going to be seeing them again in the next chapter.

We've also touched on the difficulty of correctly sorting. We had to circle back to our Unicode normalization knowledge to help create a more flexible sorting algorithm.

Knowing how to examine the contents of a string is useful but often it has another step. Modification of the string. Does the string have characters we don't want? We know how to find out. Now we need to learn how to remove them.

12 | Modification

The strings we make will likely be perfect but those other people, they're making mistakes all the time. On occasion it'll fall on us to fix these mistakes. It might also be the case that a string is good but needs a little change.

Adding

In the last chapter we talked about positions and indexes within a string. I mentioned that the accessor method had an assignment counterpart. We can use that counterpart, `[]=`, to insert characters anywhere in the string.

```
>> alpha = 'bef'
=> "bef"
>> alpha[0, 0] = 'a'
=> "a"
>> alpha
=> "abef"
>> alpha[2, 0] = 'cd'
=> "cd"
>> alpha
=> "abcdef"
>> alpha[6...6] = 'g'
=> "g"
```

```
>> alpha
=> "abcdefg"
```

What do you do if you need to add a string into the middle of another one? If you
know the position where you want the string added you can use `insert`.

```
>> 'abef'.insert(2, 'cd')
=> "abcdef"
```

You can insert text anywhere into the `'abef'` string. The positions start with 0
before the "a" all the way through 4 after the "f".
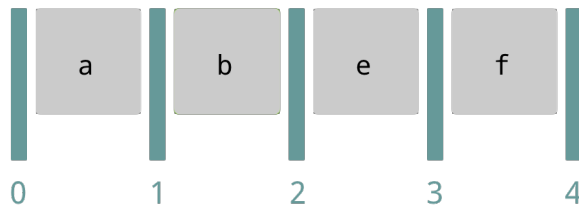
In my experience, the most common places to add string are the start and end of an-
other string. This makes `insert` take a back seat to methods like `concat`, `prepend`,
and `+`. When we do insert character mid string it's not usually positional. Often it'll
be based on some content inside the string. For insertion based on content we'll
need `sub` and `gsub`.

Both `sub` and `gsub` operate the same way. They take a regular expression used to
find a section of the string and a substitution mechanism. The only difference is
`sub` stops after the first match and `gsub` keeps going. There are several options for
how the substitution occurs. The most simple solution is to use a string.

```
>> 'abcdef'.sub(/a/, 'a*')
=> "a*bcdef"
```

Notice that we have to replace the "a" we were looking for. By repeating the "a" we've
created an opportunity to mess up. If anyone wants to change the letter they'll need

to do it in both places. In this example it's pretty obvious but it could be more subtle in a more complicated substitution. Even this easy case could be missed by someone with a lot on their mind. We can help people succeed by removing this chance to fail. It also limits the flexibility of the code. What if we want to check for any vowel instead of only "a"?

These methods allow us to reference the entire match by using a special syntax. We can reference the entire match within the substitution string with `\0`. Let's use `\0` to improve the previous example.

```
>> 'abcdef'.sub(/a/, '\0*')
=> "a*bcdef"
```

Now we can change the regular expression without having to worry about the substitution.

```
>> 'abcdef'.sub(/[aeiou]/, '\0*')
=> "a*bcdef"
```

If we switch this to a `gsub` we can mark both vowels.

```
>> 'abcdef'.gsub(/[aeiou]/, '\0*')
=> "a*bcde*f"
```

This small change has made our solution much more flexible.

Sometimes a string won't be flexible enough to do what we want. For maximum power we can use a block to augment our match. The block receives the entire match as a string and we can execute whatever code we need.

```
>> 'abcdef'.gsub(/[aeiou]/) { |m| "#{m}(#{m.capitalize})" }
=> "a(A)bcde(E)f"
```

We can also chain onto this version to do things like number our matches.

```
>> 'abcdef'.gsub(/[aeiou]/).with_index(1) { |m, i| "#{m}(#{i})" }
```

```
=> "a(1)bcde(2)f"
```

The final substitution method is a hash. The keys are matches and the values are the string to replace them with. As we'll explore later, it's a solution that is much better for replacement than addition.

Replacing

We can use `sub` and `gsub` for adding in characters because adding is nothing more than a special form of substitution. You substitute nothing for something. Or you substitute something for itself plus some other stuff. These methods really shine when we start replacing characters.

Instead of marking each vowel with "*" we could replace the vowel.

```
>> 'abcdef'.gsub(/[aeiou]/, '*')
=> "*bcd*f"
```

Earlier we used `\0` to reference the original match and add to it. If we have a match with capture groups they're made available to us in the same way by referencing the number of the capture.

```
>> 'abcdef'.gsub(/(.)(.)/, '\2\1')
=> "badcfe"
```

Here we have two capture groups that each grab one character. The substitution string `'\2\1'` inserts the second substitution and then the first substitution to flip each pair of characters.

The problem with referencing captures (or anything else) positionally is that you might add a new one and shift the order. Let's say that we have a list of U.S. phone numbers that we're reformatting.

```
>> '5554242'.gsub(/(\d{3})(\d{4})/, '\1-\2')
=> "555-4242"
```

Now we're told that we're going to get area codes with the phone numbers.

```
>> '7985554242'.gsub(/(\d{3})(\d{3})(\d{4})/, '(\1) \2-\3')
=> "(798) 555-4242"
```

We had to completely rework our substitution string to account for the new capture group. It's also not as clear as it could be. We're capturing groups of digits but what are they?

To solve this we can name our captures. We can reference a capture group by using \k<...> with the name of the group between the < and >. With capture groups our original phone number formatter looks like this.

```
>> '5554242'.gsub(
..    /(?<prefix>\d{3})(?<line_number>\d{4})/,
..    '\k<prefix>-\k<line_number>'
.. )
=> "555-4242"
```

We've labelled the two parts as the prefix and the line_number. We can now add the area code without affecting the existing captures.

```
>> '7985554242'.gsub(
..    /(?<area_code>\d{3})(?<prefix>\d{3})(?<line_number>\d{4})
   /,
..    '(\k<area_code>) \k<prefix>-\k<line_number>'
.. )
=> "(798) 555-4242"
```

We've improved the flexibility of the code by avoiding these positional references. In the process the entire line is longer and noisier. I would take that trade-off but we don't have to. By using the x flag on our regular expression we can space it out.

```
>> '7985554242'.gsub(/
..    (?<area_code>\d{3})
..    (?<prefix>\d{3})
..    (?<line_number>\d{4})
.. /x, '(\k<area_code>) \k<prefix>-\k<line_number>')
=> "(798) 555-4242"
```

The x flag tells the regular expression engine to ignore all the whitespace so you can format it in a more readable way. If you need to handle whitespace you can do that with \s to match any whitespace or [ ] (a character set containing a space) to match only space characters.

Like before, we can use a block to handle the substitutions. We're only given the match but we can still access any capture groups with their global variables. When a capture is made, it is stored in a global variable with a number corresponding to the position of the capture.

```
>> 'abcdef'.gsub(/(.)(.)/) { $2 + $1 }
=> "badcfe"
```

It's usually best to avoid global variables. Only getting the match doesn't mean we can't still do powerful transformations though. Let's translate a sentence into Pig Latin.

```
>> 'How are you doing today?'.gsub(/\w+/) do |match|
..    if match =~ /\A[aeiouy]/
..      match << 'yay'
..    else
..      first_letter = match.slice!(0)
..      if first_letter.upcase == first_letter
..        match.capitalize!
..      end
..      match << first_letter.downcase << 'ay'
..    end
.. end
=> "Owhay areyay youyay oingday odaytay?"
```

113

I briefly mentioned that hashes were another option for specifying substitutions. They're most useful for replacing specific lists of items. When we use a hash we're providing a lookup table for the replacements. The methods will take the match and try to find a matching key in the hash.

```
>> MORSE_CODES = {
..    'a' => '.-',
..    'b' => '-...',
..    ...,
..    'y' => '-.--',
..    'z' => '-..'
.. }
=> {"a"=>".-", "b"=>"-...", ...}
>> 'sos'.gsub(/./, MORSE_CODES)
=> "...---..."
```

Each character was looked up in the MORSE_CODES hash and replaced with its value. If a key can't be found or its value is nil then the match is replaced with an empty string. We can change that behavior by setting a default value or default Proc on the hash we provide. For example, we may want the program to raise an error if any non-Morse code character appears.

```
>> MORSE_CODES.default_proc = ->(_ ,char) { raise "#{char} is
    not valid" }
=> #<Proc:0x007fd38581fdc0 (lambda)>
>> 'sos!'.gsub(/./, MORSE_CODES)
RuntimeError: ! is not valid
```

Using sub and gsub is overkill for some situations. If we're only looking to replace a fixed portion of the string we can use []=.

```
>> alphabet = 'a'.upto('z').to_a.join
=> "abcdefghijklmnopqrstuvwxyz"
>> alphabet[3...-3] = '...'
=> "..."
>> alphabet
```

```
=> "abc...xyz"
```

With `[]=` we get functionality that's nearly identical to `sub!`. There are two subtle but important differences. The `sub!` method has a return value like many other `!` methods. If the string changes it will return the newly changed string. If there are no changes it will return `nil`. With `[]=` the return value will be the replacement string. If there's no match an `IndexError` will be thrown.

```
>> a = 'abcdef'
=> "abcdef"
>> a['abc'] = '*'
=> "*"
>> a
=> "*def"
>> a['xyz'] = '*'
=> IndexError: string not matched
```

We're not limited to substrings. Regular expressions are also fair game.

```
>> a = 'abcdef'
=> "abcdef"
>> a[/[aeiou]/] = '*'
=> "*"
>> a
=> "*bcdef"
```

We can use capture groups if we want to adjust the portion being replaced. The capture groups can be referenced by index or by name. With `0` we can reference the entire regular expression instead of a specific capture group.

```
def anonymize(full_name, portion = :all)
  full_name = full_name.dup
  portion = 0 if portion == :all

  full_name[/\A(?<first_name>.+) (?<last_name>.+)\z/, portion]
      = '*'
```

```
    full_name
end
```

This function takes a full name and anonymizes some or all of it. We'll `dup` the string so that we don't affect the original and then replace part or all of it with `*`.

```
>> anonymize('Bob Smith')
=> "*"
>> anonymize('Bob Smith', :first_name)
=> "* Smith"
>> anonymize('Bob Smith', :last_name)
=> "Bob *"
```

There's one final way to replace letters that is often overlooked. Short for translate, `tr` takes a character set expression and another character set expression to map the first one to.

```
>> 'hello world'.tr('o', '0')
=> "hell0 w0rld"
```

Each `o` was replaced with a `0`. We can replace two letters at once by listing both and providing matching replacement characters.

```
>> 'hello world'.tr('ol', '0|')
=> "he||0 w0r|d"
```

This time we replaced `o` with `0` *and* `l` with `|`. If we don't provide enough replacement characters, the last one will be used to fill in the rest.

```
>> 'hello world'.tr('loe', '|*')
=> "h*||* w*r|d"
```

Using `^` negates the match. Everything in the `^` is considered one replacement. We can't individually replace those characters.

```
>> 'hello world'.tr('^aeiou', '*')
=> "*e**o**o***"
```

We can also replace ranges of characters. For example, we can rotate each character forward 13 positions in the alphabet (i.e. rot13).

```
>> 'hello world'.tr('a-z', 'n-za-m')
=> "uryyb jbeyq"
```

Here we match "a" through "z" mapping the first half to "n-z" and the second half to "a-m". The result is "a" becomes "n", "b" becomes "o", and so on.

Imagine we want to generate a URL slug from a blog post title. We'll want swap non-alphabet and non-numeric characters for dashes. With `tr` we can easily accomplish it.

```
>> 'Ruby Book List: An Awesome New Addition'.tr('^a-zA-Z0-9',
   '-')
=> "Ruby-Book-List--An-Awesome-New-Addition"
```

We've substituted in our dashes and we could go ahead and use it in a URL. You might have noticed there's a `--` in our output. Any place where we have multiple replacements is going to result in a series of dashes. We can fix this by calling `tr_s` instead of `tr`. When a run of replacements occurs, `tr_s` will reduce that replacement to a single character. Instead of replacing `": "` with two dashes, it'll replace it with a single dash.

```
>> 'Ruby Book List: An Awesome New Addition'.tr_s('^a-zA-Z0-9'
   , '-')
=> "Ruby-Book-List-An-Awesome-New-Addition"
```

With `tr` and `tr_s` you can do a surprising number of these sorts of translations. It'll also run faster than an equivalent `gsub` call. Depending on what you're doing it'll be around 2x to 3x faster.

Removing

Once people have the hammer of `gsub` they tend to use it to bash everything in sight. Doing so is slower and, more complicated than need be.

A common issue when parsing data is trailing separators. We could use `gsub` to remove those trailing separators but it's a little heavy.

```
>> "This is a line of text.\r\n".gsub(/\r\n\z/, '')
=> "This is a line of text."
```

This works because we can replace the contents of the match with nothing. We could also use `chop` to do this. With `chop` the last character is removed. Oddly, it treats `\r\n` as a single character.

```
>> "This is a line of text.\r\n".chop
=> "This is a line of text."
```

Unfortunately, this isn't the safest thing to do. If the last line of our input doesn't end with `\r\n` we'll end up losing a character we wanted. For nearly every case, `chomp` is the better choice. It works like `chop` except that it only removes what you tell it to. By default it'll remove the current separator (i.e. `$/` or `$INPUT_RECORD_SEPARATOR` if you require `English`). This will get rid of `\n` and `\r\n`.

```
>> "This is a line of text.\r\n".chomp
=> "This is a line of text."
```

Now we can be certain we're only removing what we want. If the lines being parsed end with anything else we can pass that string and `chomp` will handle it.

```
>> 'abc'.chomp('c')
=> "ab"
>> 'def'.chomp('c')
=> "def"
```

Even for more complex removals, `gsub` might not be the right choice. If we're looking to scrub a string of certain characters we can do that with `delete`. Ruby's `delete` method takes one or more character set expressions and removes any character that matches all of the expressions provided.

```
>> 'Aaron'.delete('aeiouAEIOU')
=> "rn"

>> 'Aaron'.delete('aeiouAEIOU', '^a')
=> "arn"
```

Using `delete` over `gsub` better indicates your intent and it's faster.

To test out the speed difference I generated a string of random lower case letters. Then I deleted the vowels using `gsub` and `delete`. A test string with 10 characters showed `delete` to be about 4.5x faster. Move up to a string of 100 and the difference jumps to 7.5x. By 1,000 characters it's 20x faster.

Another nifty method for removing characters is `squeeze`. More than not you'll probably use it to compact whitespace within a string. Calling `squeeze` with no arguments takes any runs of the same character and reduces them to a single instance of that character. It's unusual to want to compress all runs of characters though. The common case is to pass a character or characters you want squeezed. Like `delete`, it takes one or more character set expressions. If a character matches all of the sets it'll get squeezed down.

```
>> 'hello     world'.squeeze(' ')
=> "hello world"

>> 'hello     world'.squeeze('l ')
=> "helo world"

>> 'peerless'.squeeze('a-z', '^aeiou')
=> "peerles"
```

Once again, `squeeze` finishes the job quicker than `gsub`. A string of 100 newlines is reduced by `squeeze` about 4x faster than `gsub`. Which is great if, you know, you do that sort of thing.

Even without the speed boost, these methods do a better job of indicating their intent. When you see `chomp`, `delete`, or `squeeze` you'll immediately know what's happening. With `gsub` you can't glance at a line and know what it does. You're forced to reason about a regular expression to figure it out.

Recap

The three basic types of string modification are adding, replacing, and removing characters. Our knowledge from the last chapter turned out to be useful here. Indexes, positions, and character set expressions all re-appeared here. We learned that while `sub` and `gsub` are extremely powerful there are sometimes better options. Methods like `delete` and `tr` are faster and better at indicating intention.

At this point you're a master of making your own strings and modifying those given to you. There are however times when the string given to you will be flawed. It will have been mangled somewhere along the way and it'll be your job to fix it. Next, we'll look at how to diagnose and correct these errors.

13 | Correction

When you work with encodings, things will go wrong. Learning to diagnose issues and fix them is a great skill to have in your toolbox. Wait, tool to have in your skill box? Skill to have in your skill box. That's the one.

Tofu

Does this look familiar?
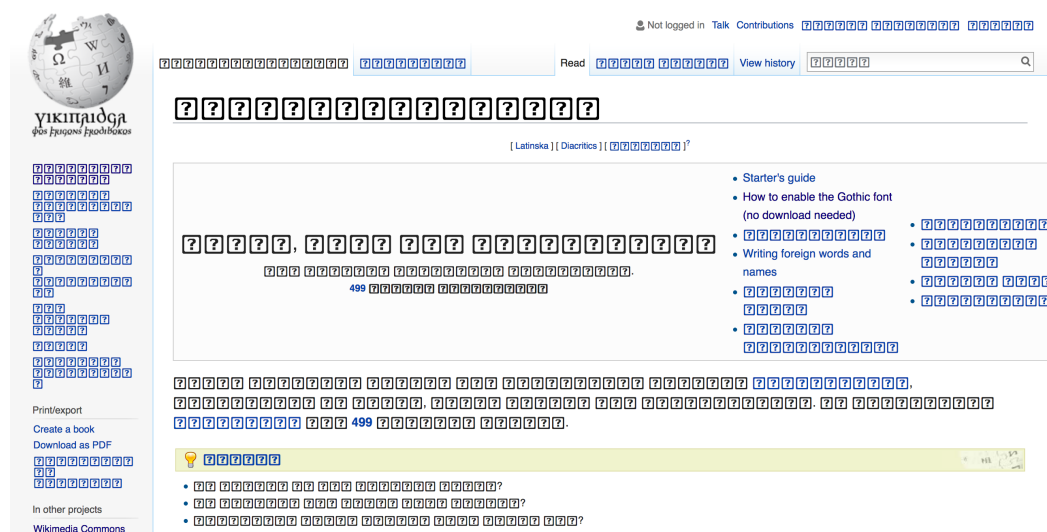
Those little square boxes can take on several forms. Sometimes they look like empty

squares. Other times they'll have a question mark or an "X" inside. Each represents a glyph that is missing from the font being used. The mark is technically a non-defined character but it's more popularly called "tofu". The nickname comes from their square, white appearance and the fact that it's more fun to say "tofu" than "bean curd".

Unicode may be prevalent but it's also massive. Almost no fonts attempt to implement every character. Beyond the sheer scale of it, the work also requires a good amount of knowledge. Font designers aim to create a cohesive look with their fonts. Understanding how every character can be styled without damaging readability is tricky work. It would take a large team with a variety of linguistic talent to create a single font that has every character. Enter Google.

Google's Noto font attempts to implement every Unicode character[1]. In fact, Noto stands for "**no** more **to**fu". All those character glyphs come at a price. Fonts usually top out at a few megabytes but Noto comes in at more than 470 MB. It's not exactly the kind of thing you drop into a webpage just to cover your bases.

Some tofu will list the code point of the missing glyph inside itself. When a lot of characters are missing it won't be practical to look them up. If it were one or two that might be different. Usually, you're be better off finding a font with proper support.

Invalid Characters

Sometimes it's not the font that's lacking, it's the string itself. Not every combination of bytes is going to produce valid characters. If a byte exists which does not represent any character in the encoding we'll get the replacement character, "�".

```
$ irb -E ISO-8859-1
>> string = ''
```

---

[1]https://www.google.com/get/noto/

```
=> ""
>> string << 0x61 # a
=> "a"
>> string << 0x62 # b
=> "ab"
>> string << 0xe7 # ç
=> "ab�"
```

What I've done here is tell my UTF-8 terminal to render an ISO-8859-1 character. The "a" and "b" work because they use the same code points for those characters. The "ç" isn't the same. ISO-8859-1 stores that as "11100111" which is invalid in UTF-8. Instead of blowing up, my terminal substitutes in the replacement character for the bad byte and continues on. The string is fine, it's the terminal that's interpreting it incorrectly.

When Ruby runs into a bad byte it prints it out as a hexadecimal code. In a normal IRB session (UTF-8) I can force the same byte pattern we saw earlier. In this case I've got the same bytes as the ISO-8859-1 version but Ruby thinks the string is UTF-8. It can't read the bad byte so it tells us what the byte is.

```
>> string = 'abc'
=> "abc"
>> string.setbyte(2, 0xe7)
=> 231
>> string
=> "ab\xE7"
```

We can check a Ruby string to make sure the bytes are valid for the encoding with valid_encoding?.

```
>> string
=> "ab\xE7"
>> string.encoding
=> #<Encoding:UTF-8>
>> string.valid_encoding?
=> false
```

From here we can force `string` to be ISO-8859-1. In that encoding the bytes we're using are valid.

```
>> string.force_encoding('ISO-8859-1')
=> "ab\xE7"
>> string.valid_encoding?
=> true
```

A more troubling version of these errors is when the bytes are valid but wrong. Let's say we pass "über" from a program using UTF-8 to another using ISO-8859-1. On the screen it then renders as "Ã¼ber". It does this because of the bytes involved. We can see that the "ü" is breaking. Everything after that looks fine.

If we break down the bytes in the UTF-8 string we see there are five.

```
>> 'über'.bytes
=> [195, 188, 98, 101, 114]
```

The "ü" is made up of `195` and `188`. ISO-8859-1 sees those bytes as distinct characters. The 195 maps to "Ã" and 188 maps to the vulgar fraction "¼". It's "vulgar" based on an old definition of the word that means "common to the masses". It's not like ¼ gets belligerent at the bar and starts making lewd references to the 3 sitting in the corner. Anyway, we can see at a glance that something's up. We can make a quick judgement about the likelihood of "Ã¼ber" being a word. Computers are less judgmental. Maybe you wanted some crazy combination of characters. The result is you'll notice it because you see it, often in production. This garbled mutant text is known as mojibake. It's a Japanese word meaning "character transform".

The problem here is mojibake still renders valid characters.

```
>> uber = [195, 188, 98, 101, 114].pack('C*')
=> "\xC3\xBCber"
>> uber.valid_encoding?
=> true
```

This was supposed to be the UTF-8 string "über" but our application accepted it as an ISO-8859-1 by mistake and Ruby can't tell us it's invalid.

Fix What You Find

Knowing is half the battle. We also have to know how to fix this mishmash. The simplest of the bunch is `scrub` and its mutating cousin `scrub!`. With `scrub` you can replace invalid bytes. By default it uses the replacement character but we can also pass in a character that we'd prefer.

```
>> string = [0x61, 0x62, 0xe7].pack('C*').force_encoding('UTF
   -8')
=> "ab\xE7"
>> string.scrub
=> "ab�"
>> string.scrub('👻')
=> "ab👻"
```

If you really want to get your hands around those bad bytes you can pass a block.

```
>> string.scrub { |char| "👉#{char.bytes}👈" }
=> "ab👉[231]👈"
```

In this case we can fix the issue by treating the bytes like UTF-8 characters.

```
>> string.scrub! { |char| char.bytes.pack('U') }
=> "abç"
>> string.bytes
=> [97, 98, 195, 167]
```

With `pack`, the `231` (i.e. `0xe7`) was treated as though we appended it to an existing UTF-8 string. Ruby knows that means converting it into the correct 2-byte UTF-8 sequence.

You will commonly see issues like this when converting between encodings.

```
>> string = [0x61, 0x62, 0xe7].pack('C*').force_encoding('UTF
   -8')
=> "ab\xE7"
>> string.encode('US-ASCII')
Encoding::InvalidByteSequenceError: incomplete "\xE7" on UTF-8
```

You could use `scrub` on this like we did earlier or you can leverage `encode` to fix the issue. Knowing that this is common, `encode` has an `:invalid` option that will let us replace the bad byte instead of throwing an error. The only options you can pass to `:invalid` are `:replace` or `nil`.

```
>> string
=> "ab\xE7"
>> string.encode('US-ASCII',
..    invalid: :replace
.. )
=> "ab?"
```

The `"\xE7"` was replaced with the replacement character. For Unicode that'll be "�" but for everything else we'll get a normal question mark. If the question mark isn't your jam it can be easily swapped out with `:replace`.

```
>> string.encode('US-ASCII',
..    invalid: :replace,
..    replace: '<Invalid>'
.. )
=> "ab<Invalid>"
```

Encoding conversion is tricky business. Characters are available in some encodings but not others. Take a word like "résumé". If we were writing that in English we'd drop the accents. That's how we like it. We prefer to have no way to distinguish between words like resume and resume.

As a result US-ASCII doesn't handle accents. So, when you try to convert "smörgås-

126

bord" into US-ASCII it doesn't go so well.

```
>> 'smörgåsbord'.encode('US-ASCII')
Encoding::UndefinedConversionError: U+00F6 from UTF-8 to US-
    ASCII
```

That's not a surprising response. It's the same thing that happened with an invalid byte. Ruby threw an error at us. We can correct this with :undef. It works exactly like :invalid except it handles undefined conversions instead of invalid bytes.

```
>> 'smörgåsbord'.encode('US-ASCII',
..    undef: :replace
.. )
=> "sm?rg?sbord"

>> 'smörgåsbord'.encode('US-ASCII',
..    undef: :replace,
..    replace: '<Undefined>'
.. )
=> "sm<Undefined>rg<Undefined>sbord"
```

If we want to take a more nuanced approach to replacement we'll have to use the :fallback option. This takes an object that has an accessor method (i.e. []) and gives it the undefined character. You **can not** use this with :undef. If you do, Ruby will ignore the :fallback option. With a hash we can map out our replacements.

```
>> 'smörgåsbord'.encode('US-ASCII',
..    fallback: { 'ö' => 'o', 'å' => 'a' }
.. )
=> "smorgasbord"
```

The hash will need to have a key for each character that you want to replace. If one is missing, it'll pass it straight through and blow up just like it did before.

```
>> 'smörgåsbord'.encode('US-ASCII',
..    fallback: { 'ö' => 'o' }
.. )
```

```
Encoding::UndefinedConversionError: U+00E5 from UTF-8 to US-
    ASCII
```

Using a default hash value can get you around this issue. Any key that can't be found
will result in the default being reported back from the hash.

```
>> MAP = { 'ö' => 'o' }
=> {"ö"=>"o"}
>> MAP.default = '?'
=> "?"
>> 'smörgåsbord'.encode('US-ASCII', fallback: MAP)
=> "smorg?sbord"
```

I mentioned earlier that `:fallback` will take any object that responds to `[]`. That
means you can also use a proc to handle the replacements. It's an uncommon way
to call a proc but you can do so with `[]`.

```
>> Proc.new { |name| puts "hello #{name}" }['reader']
hello reader
=> nil
```

The same thing works for lambdas which are just a type of `Proc`.

```
>> greet = ->(name) { puts "hello #{name}" }
=> #<Proc:0x007fbde79dcf40 (lambda)>
>> greet['reader']
hello reader
=> nil
```

Using a proc we can make `:fallback` do whatever we like. For instance, we could
decompose the character and then remove anything from the Unicode mark range.
This means an "ö" turns into "o". An "□" would become "A" and other letters would
be corrected to a more US-ASCII friendly form. If we don't get a conversion we'll
use the "?" as our invalid character identifier.

```
>> 'smörgåsbord'.encode('US-ASCII', fallback: ->(char) {
```

```
..    new_char = char.unicode_normalize(:nfd).delete("\u0300-\
   u036f")
..    if new_char != char
..      new_char
..    else
..      '?'
..    end
.. })
=> "smorgasbord"
```

If you're encoding the text for use in XML (or even HTML), you might like the `:xml` option. This option lets you ready a string to be used within an XML document as either text or an attribute. If we pass `:text` it'll replace undefined characters with their hexadecimal references. It also replaces "&", ">", and "<" with their escaped forms.

```
>> 'smörgåsbord > cornucopia'.encode('US-ASCII', xml: :text)
=> "sm&#xF6;rg&#xE5;sbord &gt; cornucopia"
```

The "ö" was replaced with `ö`, "å" became `å`, and ">" was swapped with `>`. If we work with XML style attributes we can use `:attr` instead. In addition to what `:text` does, it also escapes double quotes with `&quote;` and surrounds the output in double quotes.

```
>> %q(He screamed, "Duck!").encode('US-ASCII', xml: :attr)
=> "\"He screamed, &quot;Duck!&quot;\""
```

There's one more transform that you can do with `encode`: newlines. They haven't always been consistent between operating systems. The `encode` method comes with three options to help resolve newline issues. There's `:cr_newline` which replaces "\n" with "\r", `:crlf_newline` which replaces "\n" with "\r\n", and `:universal_newline` that replaces both "\r" and "\r\n" with "\n".

```
>> "\n".encode('UTF-8', cr_newline: true)
=> "\r"
```

```
>> "\n".encode('UTF-8', crlf_newline: true)
=> "\r\n"

>> "\r \r\n".encode('UTF-8', universal_newline: true)
=> "\n \n"
```

In these cases we didn't even change the encoding. We only leverage `encode` for its ability to change the type of newline. These options are also made available on `IO#open` and `File#open` so you can do all of this while reading in a file.

Deeper Into the Abyss

If you ever find yourself needing more power than `encode` provides you should probably run. Assuming that's not an option and, you're all out of genie wishes, you'll want to use `Encoding::Converter`. It lets you get down and dirty into the conversion process.

When converting between encodings, Ruby may have to make a few jumps to get from a to z. For example, going from ISO-8859-1 to ISO-8859-2 it uses UTF-8 as an intermediate. Rather than write every permutation of one encoding to another, Ruby can write converters to UTF-8 and leverage it in the middle. We can get a complete list of the steps between any two encodings with `search_convpath`.

```
>> Encoding::Converter.search_convpath('ISO-8859-1', 'ISO
   -8859-2')
=> [
    [#<Encoding:ISO-8859-1>, #<Encoding:UTF-8>],
    [#<Encoding:UTF-8>, #<Encoding:ISO-8859-2>]
   ]
```

Most paths are no more than a hop or two. Converting between ASCII and ISO-

130

2022-JP ends up using four hops to get the job done.

```
>> Encoding::Converter.search_convpath('ASCII', 'ISO-2022-JP')
=> [
     [#<Encoding:US-ASCII>, #<Encoding:UTF-8>],
     [#<Encoding:UTF-8>, #<Encoding:EUC-JP>],
     [#<Encoding:EUC-JP>, #<Encoding:stateless-ISO-2022-JP>],
     [#<Encoding:stateless-ISO-2022-JP>, #<Encoding:ISO-2022-
        JP (dummy)>]
   ]
```

With `Encoding::Converter` we can also create a new converter. Then we can encode strings as we go. It'll take a source encoding for input strings, a destination encoding for the output string, and almost all of the same options as `encode`. The only one it doesn't accept is `:fallback`. After creating the converter we pass strings in via `convert`.

```
>> converter = Encoding::Converter.new('UTF-8', 'US-ASCII',
..    undef: :replace
.. )
=> #<Encoding::Converter: UTF-8 to US-ASCII>
>> converter.convert('smörgåsbord')
=> "sm?rg?sbord"
```

The real power of `Encoding::Converter` lies in `primitive_convert`. This method takes a source string, a destination string, and if you want to get really specific, some byte offset and size options.

```
>> converter = Encoding::Converter.new('UTF-8', 'US-ASCII')
=> #<Encoding::Converter: UTF-8 to US-ASCII>
>> src = 'smörgåsbord'
=> "smörgåsbord"
>> dst = ''
=> ""
>> converter.primitive_convert(src, dst)
=> :undefined_conversion
```

The `primitive_convert` method does its job until it bumps into a problem. It'll report the problem via a symbol key or return `:finished` if everything went well. Here we got an `:undefined_conversion` which means it encountered a character that it couldn't represent. As characters are converted they're moved from `src` to `dst`. We can see exactly where the problem was by looking at the current state of those strings.

```
>> src
=> "rgåsbord"
>> dst
=> "sm"
```

We can also get details about the error by checking in with `last_error` and `error_char`.

```
>> error = converter.last_error
=> #<Encoding::UndefinedConversionError: U+00F6 from UTF-8 to
   US-ASCII>
>> error.error_char
=> "ö"
```

There are a few other methods that let you insert some output into the destination and put bytes back into the source. They're methods you'll probably never need.

Recap

When we find invalid characters we can fix them with `scrub` or `encoding`. We learned how to trace the path of conversions that lead from one encoding to another. If things go really wrong we've seen that we can deep dive with `Encoding::Converter`.

14 | Strings!

Before this you'd likely only seen the part of the iceberg sticking out above the water. The icy crest that looks grand but reveals only a tenth of the reality. Now, together, we've dove down and explored the monster in its entirety.

One layer at a time we've gone from bit to character to character set and beyond. We've turned the String object inside-out and examined it from every angle. I'll bet you can tell me the difference between `%q` and `%Q` and why `'à'.size` might return `2`. If you've forgotten, that's ok too. It's a lot to take in.

Don't hesitate to re-open this book and skim or read a section again. It's been organized so topics are easy to find and return to. Glance over the table of contents and you'll likely see what you're looking for. My hope is that you'll keep this handy as a guide and reference it from time to time.

In the beginning, I said you'd know more about strings than you thought possible. I hope that I've shown you the value not only of strings but of intimately knowing a tool. Take a moment and list the ones you use the most. Think about objects, data structures, gems, command line utilities, and whatever else you regularly use to get your work done. Consider the benefits you'll uncover if you take it upon yourself to explore below the surface. We've covered strings together. Now it's your turn. What will you pick for your next deep dive?

Hi, I'm Aaron Lasseigne. I've been programming in Ruby for more than a decade. Sharing what I learn has always been a part of what I do. I've volunteered for Rails Girls, presented to high school students, and mentored Rubyists through the Dallas Ruby Brigade which I help organize. My writings have appeared in numerous Ruby Weekly issues as well as a variety of other sites. I also contribute to open source. My most successful project is the ActiveInteraction gem with over 900 stars on GitHub.