2 The King and his String

a short yarn

The King was in a foul mood. I mean a *truly terrible*, scream-at-the-cat, throw-a-snowglobe-out-a-third-story-window kind of mood. If you saw him rumbling toward you down the sidewalk, you would quickly change sides of the street. If he were your dad, you would write letters to Santa year-round asking for a replacement dad. Really, it was capital-*B* Bad News Bears for everyone.

The thing is, the King had lost his favorite possession that morning, somewhere between eating his usual breakfast of parched oats and his pre-late-afternoon vigorous stroll. He had turned his palace upside-down (literally: the King had a lot of money and a lot of servants), but to no avail. When Scarlet and Ruben found him, he was weeping bitterly in his study, sitting in an overstuffed armchair of solid gold.

“What did it look like?” Scarlet asked.

“What did *what* look like?” asked the King, gargling slightly on his own salty tears as they flowed down his finely coiffed moustache and into his mouth.

“The thing you lost,” said Ruben.

“Like a string!” said the King. “Because that’s what it was: a string, with a knot on each end to secure my bits and trinkets. This particular string had several beads on it that spelled out ‘Property of His Royal Highness, the King,’ like so:

```
'Property of His Royal Highness, the King'
```

“A string of letters,” said Ruben.

“More like a string of *characters*,” said the King. “Each letter is really very unique. The *K*, for instance, is a crooked fellow. And don’t even get me started on the *p*—”

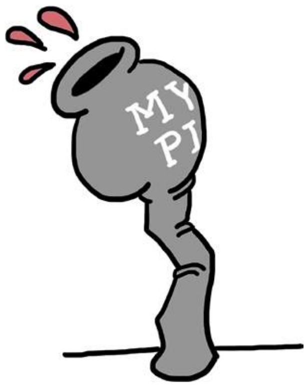
Property of
His Royal
HigHNES's,
the King

But Ruben and Scarlet weren't listening. They were already searching high and low for the King's missing string.

"Could your string have fallen into this Mysterious Pipe?" Scarlet asked, gesturing toward a sputtering black metal pipe with the words *Mysterious Pipe* written on it in white chalk.

"No," said the King. "The Mysterious Pipe is deceptively narrow at the top, and a string as long as mine could never fit into it."

"How long *is* your string?" Ruben asked.



"I'm not sure," said the King. "I suppose we could count all the characters, and then we'd know." (Take it from me: this would be super boring.)

"That would be boring," said Scarlet. "I think there's a better way." She walked to a corner of the room, blew the dust off a very old Computing Contraption, and carefully typed the following at its little green IRB prompt:

```
>> 'Property of His Royal Highness, the King'.length
=> 40
```

"Great coats!" said the King. "That's right! I remember now—my string is precisely 40 characters long. But how did you do that?"

"Ruby has lots of great tricks like this," said Scarlet. "Here's another."

```
>> 'Property of His Royal Highness, the King'.reverse
=> "gniK eht ,ssenhgiH layoR siH fo ytreporP"
```

The King nodded. "Yes, that's pretty much what my string looks like in the mirror when I hang it up to dry after a refreshing shower."

In the meantime, Ruben had been counting the number of characters in the King's string using a bit of chalk he found resting near the Mysterious Pipe. "Hang on a second," he said. "I'm counting 42 characters, including the quotation marks on each end."

The King snorted like an overweight wiener dog. "You don't count those!" he said. "Those are the little knots on each end that keep the characters contained! You only count the characters, not the quotes."

"And that's exactly what Ruby does," explained Scarlet. "But you have to put quotes around your strings, or Ruby will think you're trying to use a *variable*."

A Bit More About Variables

Believe you me, this confused the bejeepers out of the King. Since he's not nearly as bright a bulb as you are, I'll let Ruben and Scarlet spend ages explaining variables to him while I take a moment to explain them to you.

A Ruby variable is just a name (without quotes!) that you can give to a *value* (which is a piece of information, like the words that make up the King's string). One kind of value is a string; another kind is a *number*, which you already saw when Ruby told you that the length of the King's string was 40.

You make a variable like this:

```
>> kings_string = 'A string fit for a king'
>> wiener_dog_weight = 22
```

The equal sign says to Ruby, "Hey! Take this value on the right and save it with the name on the left." This means that later on, you can type the variable name and get the value right back:

```
>> wiener_dog_weight
=> 22
```

This could come in handy when you're trying to keep track of your wayward pet (let's call him Smalls) and his fluctuating weight:

```
>> smalls_weight = 22
=> 22
>> pounds_lost = 4
=> 4
>> smalls_new_weight = smalls_weight - pounds_lost
=> 18
```

Don't worry about the 22 and the 4 being repeated back to you; Ruby's just trying to be helpful. Ruby always expects the variable name to be on the left and the value to be on the right, so make sure not to mix up the order!

You'll also notice I used _ (called an *underscore*) instead of a space in the variable names. Ruby doesn't allow spaces in names, so it's a good practice to use _ instead.

It sounds like the King is still getting the hang of strings (imagine my ear pressed to the heavy oak door of his study), so I'll clue you in on one more bit of Ruby magic. When you see code like this:

```
>> 'Property of His Royal Highness, the King'.reverse
```

it means you're *calling* the reverse *method* on the string. When we say we're "calling a method," what we mean is we're asking Ruby to carry out a command: "Hey, Ruby! Reverse this string for me, please!" I'll go on and on about methods later, but for now, you can think of them as commands that work on particular Ruby objects. For example, strings can be reversed, but numbers can't:

```
>> "18".reverse
=> "81"
>> 18.reverse
=> NoMethodError: undefined method `reverse' for 18:Fixnum
```

`NoMethodError`!? That's Ruby saying, "Whoa, whoa, *whoa*. I know how to reverse a string, but I don't know how to reverse a number!" As you practice, you'll get to know which methods go with which kinds of Ruby objects. Author's honor. (I was never a scout.)

Ruby Operators

"Let me see if I've got this right," said the King. "Variables are names for Ruby values, like strings and numbers. They don't have quotes around them and can't have spaces in them. I can use the equal sign to set a variable equal to a value, and then I can use my variable's name to get that value back."

"That's exactly right," said Ruben.

"And when I see an object followed by a dot followed by a command, that means I'm using that command on that object," said the King.

"Precisely," said Scarlet.

"You mentioned that I can't reverse a number," said the King. "That makes sense. But what *can* I do to a number?"

"All sorts of things," said Ruben. He nudged Scarlet aside and typed at the Computing Contraption:

```
>> 100 + 17
=> 117
>> 50 - 20
=> 30
>> 10 * 10
=> 100
>> 40 / 20
=> 2
```

"Yes, yes," said the King. "I can add them with `+`, subtract them with `-`, multiply them with `*`, and divide them with `/`."

"You've probably seen ÷ for division," Ruben continued, "but in code we can just use `/`. For example, 4 ÷ 2 will be `4 / 2`."

"But what can I do that's *interesting*?" the King complained.

"What about this?" asked Ruben, as he typed some more.

```
>> 22.next
=> 23
>> 22.pred
=> 21
```

"Aha!" said the King. "*Now* you're talking. `next` must tell Ruby to calculate the *next* number, and `pred` asks Ruby for its *predecessor*, which is the number that comes right before it."

"Right as rain," said Ruben.

"RAIN!" exclaimed the King, jumping up so forcefully that he knocked his solid gold armchair right over. He ran out of the room at what seemed an impossible speed for a man of his age, and Ruben and Scarlet followed.

After running for several minutes through the horribly jumbled contents of the palace (the King had turned it upside-down, after all), Ruben and Scarlet caught up with the King in his main bathroom. He was weeping again, but this time with joy, and clutched in his hands was—his string!

"Rain reminded me that I took a refreshing shower after my breakfast of parched oats!" blubbered the King. "And here it was, hanging to dry, just as I'd left it. I can't thank you enough!"

"Careful!" said Scarlet. "Your string's still a bit wet; look at the beads sliding around on it."

The King sniffed loudly and inspected his string, and the characters on it were, in fact, sliding every which way. The King thought for a moment, then double-knotted each end of the string to keep his characters from sliding off:

```
"Property of His Royal Highness, the King"
```

"Double quotes!" said Scarlet. "Can you use those with Ruby strings?"

"Definitely," said Ruben, "and single- and double-quoted strings work almost exactly the same way." He pried open the King's medicine cabinet to expose a slightly-less-old Computing Contraption, then typed the following:

```
>> double_quotes = "A string's the thing"
=> "A string's the thing"
>> single_quotes = 'for a springly King'
=> "for a springly King"
```

"See?" said Ruben. "Even when we type single quotes, Ruby repeats double quotes back to us. Both work!"

"Though I think I've heard tell," said the King, "that you can put more complicated bits and trinkets in a double-quoted string than a single-quoted one."

"That's true," said Ruben, "but we'll get to that in good time." And he closed the King's medicine cabinet with a gold-plated *click*.

a Smallish Project for you

Now that you know a bit about strings, numbers, and variables, let's put together a small project: writing a program to reflect and echo the King's string. A *reflection* of something is just that thing backward, so you've probably already guessed that we'll be reverse-ing some strings. On the other hand, an *echo* of something is just that thing repeated a few times, and we'll soon see a way to repeat a string very quickly and easily. You'll weep with joy at how simple and easy it is. You'll tear out the pages of this book and use them to dry your tears.

note *For some of the longer code examples, we'll write Ruby scripts instead of using IRB! Whenever you see a filename in italics above the code, like* kings_string.rb *for the next example, that means you can write the code as a file with the given name and run it using the* `ruby` *command. Peek back at Chapter 1 if you don't remember how to do this, or ask the nearest adult to help you. You can download all the scripts that appear in this book at* http://nostarch.com/rubywizardry/. *(But remember, if you're learning to program, try typing things out yourself instead of just reading and running the code!)*

Go ahead and make a new file called ***kings_string.rb***. Then, open your file and type the following. We're going to make a short program that shows off the cool things you can do by assigning variables and how Ruby can play with strings.

kings_string.rb

```
kings_string = "Property of His Royal Highness, the King"
string_reflection = kings_string.reverse
times_to_echo = 3
string_echo = kings_string * times_to_echo
puts kings_string
puts string_reflection
puts string_echo
```

The first four lines are assigning variables. You can tell by the equal sign.

The second line in particular is pretty cool: it defines a variable to hold the kings_string, but because the reverse method makes the string backward, string_reflection will actually be "gniK eht ,ssenhgiH layoR siH fo ytreporP"!

You might be wondering about the fourth line of code, too:

```
string_echo = kings_string * times_to_echo
```

And you're right to wonder! The * is the Ruby way of saying "multiply by." This means 2 * 2 would equal 4, 13 * 379 would equal 4,927, and so on. *But wait!* you might further wonder, *How can you multiply a string (which is just a bunch of letters) by a number?* The answer is that Ruby is quite the clever robot. When it sees something like this:

```
>> "Hello!" * 3
```

it does this:

```
=> "Hello!Hello!Hello!"
```

So this is how we produce our echo: kings_string * times_to_echo will become "Property of His Royal Highness, the King" repeated three times!

puts is short for "put string," as in "Put that string on the table where I can see it." As we've seen, it just prints text on the screen. What do you think you'll see when you run your program? Save and close your file, and then run it with **ruby kings_string.rb**. You should see the following output:

```
Property of His Royal Highness, the King
gniK eht ,ssenhgiH layoR siH fo ytreporP
Property of His Royal Highness, the KingProperty of His Royal
Highness, the KingProperty of His Royal Highness, the King
```

Well done!

You Know This!

Let's take a minute to review all the stuff you've packed into your brain over the last few pages.

We talked about *strings* and how they're just words or phrases between quotes (single or double quotes are both fine). In fact, since the bits that make up a string don't have to be just letters—they can include punctuation and even numbers, so long as the whole string is between quotes—we say that strings are made up of *characters* rather than letters. You can think of a string as a *literal* string of characters, with each end knotted with either single or double quotes. (You can pick single or double, but the ends have to match: `"string'` or `'string"` won't work!)

You also saw that strings have some handy *methods*, like `length` and `reverse`, which are just commands that Ruby knows how to use with strings. You always write the object you want to affect, followed by a dot, followed by the command, like this:

```
"gadzooks".length
```

We talked a bit about *numbers*, which are values in Ruby that work exactly like you think real-life numbers would. Numbers have their very own methods, which include `next` (for going to the next number) and `pred` (for going to the previous number):

```
>> 4.next
=> 5
```

Last, we talked about *variables* and how you can use them to give Ruby values special names, like `42` or `"chunky bacon"`. You always write the variable name (which can't contain spaces) on the left, followed by an equal sign, followed by the value:

```
>> bacon_consistency = "chunky"
=> "chunky"
>> number_of_bacon_strips = 3
=> 3
```

And you can get that value back just by typing its name:

```
>> bacon_consistency
=> "chunky"
```

Given what you know, how could you go further with that smallish project we tackled earlier? For instance, what if we changed the number of `times_to_echo` with `next` or `pred`? What would happen if we added a space on the end of the sentence we stored in `kings_string`? (Hint: It might make our output look nicer. But don't put the space directly on the variable name `kings_string`—remember, Ruby variable names can't have spaces!) What happens if we try to add a few different strings together with `+` instead of multiplying them by a number? And what in breakfast's good name is chunky bacon, anyway?