Encryptor

Have you ever wanted to send a secret message?

The Plan

Today we use computers to mask messages every day. When you connect to a website that uses "https" in the address, it is running a special encoding on your transmissions that makes it very difficult for anyone to listen in between you and the server.

The science of masking and unmasking data is called *cryptography*. When you take readable data and mask it, we say that you're "encrypting" the data. When you turn it back into readable text, you're "decrypting" the data.

Let's build a tool named "Encryptor" that can take in our messages, encrypt them for transmission, then decrypt messages we get from others.

Encryption Algorithms

An *algorithm* is a series of steps used to create an outcome. For instance, a recipe is a kind of algorithm – you follow steps and, hopefully, create some food.

There are many algorithms used in cryptography. One of the easiest goes back to the days of Ancient Rome.

Julius Caesar needed to send written instructions from his base in Rome to his soldiers thousands of miles away. What if the messenger was captured or killed? The enemy could read his plans!

Caesar's army used an algorithm called "ROT-13".

Building a Cipher

"ROT-13" is an algorithm that uses a cipher. A cipher is a tool which translates one piece of data to another. If you've ever used a "decoder ring", that's a cipher. The cipher has an input and an output.

Let's make a cipher for "ROT-13":

1. Take a piece of lined paper
2. In a column, write the letters A through Z in order down the left side.
3. On the first line, start a second column by writing the letter N next to your A
4. Continue down the alphabet, so you now write "O" to the right of "B"
5. When your second column gets to "Z", start over again with "A"

The left side of your cipher is the input, the right side is the output.

Using the Cipher

You take each letter of your secret data, find it in the left column, and write down the letter on the right. Now you have an encrypted message.

To decrypt a secret message, find the letter on the right side and write down the letter on the left.

Exercises

1. What is the result when you encrypt the phrase "Hello, World"?
2. What is the decrypted version of a message "anqn"?

Starting Encryptor

The Big Picture

We need an object which will perform the encryption and decryption operations. We will define a class then write methods in that class. From IRB we can load that class, create an instance of it, then tell it to do the work with our messages.

Setting up the Class

We'll create a class which does both the encrypting and decrypting. Let's start it off like this:

```ruby
class Encryptor

end
```

And save it in a file named `encryptor.rb`

Checking File Location

Go to your Command Prompt window and make sure it's currently in the same directory as your `encryptor.rb` file.

On Windows

From the command prompt, run this instruction:

Terminal

```
$ dir
  Volume in drive C has no label
  Volume Serial Number is 3C53-24A5
  Directory of C:\Users\YOURUSERNAME
  11/06/2012 12:00 <DIR> .
  11/06/2012 12:00 <DIR> ..
```

```
11/08/2012 12:00 <DIR> encryptor.rb
...
```

And it will output the files in the current directory. In that list you should see `encryptor.rb`.

If you *don't* see it, first make sure you saved the file in your editor. Then use `cd` to change to the correct directory.

On MacOS

From the command prompt, run this instruction:

Terminal

```
$ ls
  Desktop Downloads Movies Pictures encryptor.rb
  Documents Library Music Public
```

And it will output the files in the current directory. In that list you should see `encryptor.rb`.

If you *don't* see it, first make sure you saved the file in your editor. Then use `cd` to change to the correct directory.

Loading it in IRB

Now that you know you're in the right directory, run IRB:

Terminal

```
$ irb
  1.9.3p194 :001 >
```

Then, at the IRB prompt, tell it to load your file:

IRB

```
2.1.1 :001> load './encryptor.rb'
            => true
```

Creating an Instance

Now that IRB has loaded your code, you can create an instance of the `Encryptor` class like this:

IRB

```
2.1.1 :001>  e = Encryptor.new
             => #<Encryptor:0x007f7f39060440>
```

The second line there is the output you'll see in IRB. The numbers/letters at the end will be different.

Writing an `encrypt` Method

We made a class, that's a start – but it doesn't **do** anything!

We need to write an `encrypt` method.

The Simplest Thing That Could Work

In computer programming, you want to first do the simplest thing that could possibly work. Often this is a solution that's difficult to maintain as requirements change or somehow "cheating." But the idea is that you get it *working* first, then you make it *better*.

Let's build a really simple implementation of `encrypt` using a lookup cipher like you made on paper.

Lookup Tables

When you created the cipher on paper you made what's called a "lookup table". It's a tool which you use by having some value, finding it in the list, and getting out some related value.

For instance, when the *input* to your lookup table is `"A"`, then the output is `"N"`. Let's build a lookup table in Ruby.

The easiest way to build a lookup table is to use a Ruby *hash*.

A Quick Reminder about Hashes

If you recall, a hash is a collection of *key-value pairs*. When you want to find data in a hash, you give it the *key* and it gives you back the matching *value*.

Try this in IRB:

IRB

```
2.1.1 :001>  sample = {"name" => "Jeff", "age" => 12}
             => {"name"=>"Jeff", "age"=>12}
```

On the right side I've created a hash by using the `{` and `}`. Inside those curly brackets, I created two key-value pairs. The first one has the key `"name"` which points to a value `"Jeff"`. Then a comma separates the first pair from the second pair, then the key `"age"` points to the value `12`.

I can lookup values in that hash like this:

IRB

```
2.1.1 :001> sample["age"]
            => 12
2.1.1 :002> sample["name"]
            => "Jeff"
```

Building the Cipher / Lookup Table

This part is going to be boring. That's common when we're building "the simplest thing that could possibly work"!

We need to write out a hash which has every letter of the alphabet and which letter it corresponds to in the cipher. Use your paper cipher as a guide. Here's how it starts within your `encryptor.rb`:

```ruby
class Encryptor
  def cipher
    {'a' => 'n', 'b' => 'o', 'c' => 'p', 'd' => 'q',
     'e' => 'r', 'f' => 's', 'g' => 't', 'h' => 'u',
     'i' => 'v', 'j' => 'w', 'k' => 'x', 'l' => 'y',
     'm' => 'z', 'n' => 'a', 'o' => 'b', 'p' => 'c',
     'q' => 'd', 'r' => 'e', 's' => 'f', 't' => 'g',
     'u' => 'h', 'v' => 'i', 'w' => 'j', 'x' => 'k',
     'y' => 'l', 'z' => 'm'}
  end
end
```

Make sure you use all *lowercase letters*. If you neatly line up your rows like I did, it will make it a lot easier to find any typos, missing quotes, or missing commas.

Encrypting One Letter

Whew, that was tiring! Let's see if it actually worked.

We'll need to write an `encrypt` method. It'll take in just one letter and give us back the corresponding letter from the cipher.

Add this **inside** your `Encryptor` class:

```ruby
def encrypt(letter)
  cipher[letter]
end
```

Try It Out

In your IRB window, run these instructions:

IRB

```
2.1.1 :001> load './encryptor.rb'
            => true
2.1.1 :002> e = Encryptor.new
            => #<Encryptor:0x007f7f39060440>
2.1.1 :003> e.encrypt('m')
            => 'z'
```

Did it give you back `'z'`? If not, look for typos in your code or lookup table.

A Little Issue

In the same IRB window, try this:

IRB

```
2.1.1 :001> e.encrypt('M')
```

Did you get back `nil`? Ugh! Remember that `nil` means "nothing" in Ruby. Our `encrypt` method is looking in the `cipher` for a key `"M"`, but it isn't finding it. As far as Ruby is concerned, `"M"` and `"m"` are totally different things.

What should we do? Add all the capital letters to our cipher? That'll take us another 10 minutes!!!

A Hack to Deal with Uppercase/Lowercase

Let's implement a hack (or a "cheat") and say that no matter whether the incoming letter is uppercase or lowercase, we're going to output lowercase. Spies who are using our advanced cryptography system can fix the letters to uppercase on their own!

What we'll do here is modify the `encrypt` method so it turns *every* input into lowercase *before* looking it up in the cipher. If you pass in `"a"` it will just stay as `"a"` and be found in the cipher. If you pass in `"A"` it will be changed to `"a"` and found in the cipher.

Let's change our `encrypt` method to use the `.downcase` method. This method turns any string into all lowercase letters.

```ruby
def encrypt(letter)
  lowercase_letter = letter.downcase
  cipher[lowercase_letter]
end
```

Then go to IRB and try it again:

IRB

```
2.1.1 :001>  load './encryptor.rb'
             => true
2.1.1 :002>  e = Encryptor.new
             => #<Encryptor:0x007f7f39060440>
2.1.1 :003>  e.encrypt('M')
             => 'z'
```

If it gives you back `"z"`, then everything is on track!

Encrypting a Whole String

Our program is really sweet, as long as your message is only one letter and you don't need to ever decrypt it. Hmmmm....

An Experiment

Maybe it will magically work! Let's try this in IRB:

IRB

```
2.1.1 :001>  e.encrypt("Hello")
```

What did you get? Ugh. There's no magic in programming. We'll need to write more instructions.

A Theory

I know, let's just make a lookup table that has all the words in the English language and points to their encrypted version! Come back in 20 years when you're done typing that up.

Instead, let's encrypt one letter at a time. The *algorithm* will go like this:

1. Cut the input string into letters
2. Encrypt those letters one at a time, gathering the results
3. Join the results back together in one string

Let's do it!

Remember `.split`?

We looked at the `.split` method on Strings before. It can cut up strings like this:

```
IRB

2.1.1 :001>  sample = "Hello World"
             => "Hello World"
2.1.1 :002>  sample.split
             => ["Hello", "World"]
```

When we don’t pass in any parameters, `split` will cut the string wherever it sees a space.

What if we pass in a parameter? Try this:

```
IRB

2.1.1 :001>  sample.split("o")
```

The output was pretty different. It cut the string wherever it found an `"o"`.

Splitting Into Letters

So how does this help our `Encryptor`? We can actually pass the parameter `""` to `split`. Try this out:

```
IRB

2.1.1 :001>  sample.split("")
```

You should see an array of all the letters chopped into their own strings.

One Letter at a Time

Our existing `encrypt` method really just encrypts one letter. Let’s *change its name* to `encrypt_letter` like this:

```ruby
def encrypt_letter(letter)
  lowercase_letter = letter.downcase
  cipher[lowercase_letter]
end
```

A New `encrypt`

Then let’s start a new, blank `encrypt` method:

```ruby
def encrypt(string)

end
```

When I'm writing a program that's a little complicated, it helps me to first write **pseudocode**. Pseudocode is English that's "shaped" like code. It's a way to chart out the idea of what you want the code to do before you write it.

Taking the ideas from above, we can write pseudocode in our `encrypt` method using comments. In Ruby, any line that starts with a `#` is ignored. So here's our "blank" `encrypt` with some pseudocode:

```ruby
def encrypt(string)
  # 1. Cut the input string into letters
  # 2. Encrypt those letters one at a time, gathering the results
  # 3. Join the results back together in one string
end
```

Let's tackle those one by one.

Cut the Input String

This part we know already. We can use split like this:

```ruby
letters = string.split("")
```

Now we've got an Array of letters.

Encrypt Those Letters

This is the tricky part. Let me show you the simplest way first, then the best way second.

Gathering Results with an Array

Imagine you had a bunch of math problems to solve and needed to turn in a list of the solution. What would you do?

Probably grab a piece of paper, do the problems one by one, and write down each answer on the paper as you finish the calculation. Right?

We can do that in Ruby, too. Let's experiment in IRB:

IRB

```
2.1.1 :001> results = []
2.1.1 :002> results.push(6)
2.1.1 :003> results.push(2)
2.1.1 :004> results.push(9)
2.1.1 :005> results
```

The first line creates a blank array named `results`. The next three lines `push` a value on to the end of that array. It's like adding a value to the end of your piece of paper. Then the last line is just there to show us the values in results.

Let's use this technique in `encrypt`:

```ruby
def encrypt(string)
  # 1. Cut the input string into letters
  letters = string.split("")

  # 2. Encrypt those letters one at a time, gathering the results
  results = []
  letters.each do |letter|
    encrypted_letter = encrypt_letter(letter)
    results.push(encrypted_letter)
  end

  # 3. Join the results back together in one string
end
```

Joining the Results

`results` holds an array of letters, but we want to finish with a single string. Remember how to mash all the elements of an array together into a string?

We need the `.join` method. Call `.join` on the results array as the last line of `.encrypt`.

Testing

Try this in IRB:

IRB

```
2.1.1 :001> load './encryptor.rb'
            => true
2.1.1 :002> e = Encryptor.new
            => #<Encryptor:0x007f7f3913a3e8>
2.1.1 :003> e.encrypt("Hello")
            => "uryyb"
```

Did yours work? If you didn't get the exact same output `"uryyb"` go back and figure out what's going wrong.

Refactoring

Whenever we program we want to do the simplest thing that could possibly work. But then once it works, we try to make the code better. We can make it better by making it shorter, easier to understand, or faster to run.

This process is called **refactoring**.

My `encrypt` method currently looks like this:

```ruby
def encrypt(string)
  # 1. Cut the input string into letters
  letters = string.split("")

  # 2. Encrypt those letters one at a time, gathering the results
  results = []
  letters.each do |letter|
    encrypted_letter = encrypt_letter(letter)
    results.push(encrypted_letter)
  end

  # 3. Join the results back together in one string
  results.join
end
```

I’d first remove all the comments. The code works, so I don’t need the English words telling me what it does.

```ruby
def encrypt(string)
  letters = string.split("")

  results = []
  letters.each do |letter|
    encrypted_letter = encrypt_letter(letter)
    results.push(encrypted_letter)
  end

  results.join
end
```

The `.collect` method

The next piece I’m interested in is the middle section. Arrays in Ruby have a method named `.collect`.

The `.each` method that we’re already using goes through the elements in the array and runs the block once for each element.

The `.collect` method does the same thing, *but* it also gathers the results of running the block into an array and gives you back that array.

Let’s try an example in IRB:

IRB

```
2.1.1 :001> sample = ["a", "b", "c"]
```

```
             => ["a", "b", "c"]
2.1.1 :002>  sample.each do |letter|
2.1.1 :003>  letter.upcase
2.1.1 :004>  end
             => ["a", "b", "c"]
2.1.1 :005>  sample.collect do |letter|
2.1.1 :006>  letter.upcase
2.1.1 :007>  end
             => ["A", "B", "C"]
```

Do you see the difference in the outputs? When we use `.each`, we get back the original sample lettters. It's as though the `.upcase` never happened.

When we use `.collect` though, we get back the three letters capitalized. This array is the result of running the `.upcase` method and gathering the results.

To take it a step further, we could save the capitalized letters into a new array like this:

IRB

```
2.1.1 :001>  capitals = sample.collect do |letter|
2.1.1 :002>  letter.upcase
2.1.1 :003>  end
             => ["A", "B", "C"]
2.1.1 :004>  capitals
             => ["A", "B", "C"]
```

Now we have an array named `capitals` which holds the same results.

Challenge: Using `.collect` in `.encrypt`

Now look at your code for `encrypt`. How can you use `.collect` instead of `.each` and get rid of *two* lines of code?

Make sure that your `encrypt` method still works by testing it in IRB after you make the changes.

Decrypting

Encrypting is cool, but only if you can eventually decrypt the message.

There's this funny thing about decrypting ROT-13. There are 26 letters in the alphabet. Moving forward 13 letters is the same as moving backward 13 letters.

Write your own '`.decrypt`' method so that when tested you get output like this:

IRB

```
2.1.1 :001> load './encryptor.rb'
            => true
2.1.1 :002> e.encrypt("Secrets")
            => "frpergf"
2.1.1 :003> e.decrypt("frpergf")
            => "secrets"
```

Now you have a proper encryption/decryption tool.

Supporting More Ciphers

What if the enemy figures out the cipher?

We could change the cipher at any time. For instance, we could decide to use a "ROT-8" and only rotate eight letters.

How would you do this given the current implementation? You’d have to retype the entire cipher - yuk.

Worse, what if you want your one encryption engine to support both ROT-13 and ROT-8? What about ROT-4? ROT-20? You might need to write 26 different ciphers.

That’s ridiculous. Instead, let’s figure out how to do our encryption and decryption automatically allowing us to get rid of the original cipher all together.

Ranges

Ruby has the ability to specify ranges of letters and numbers. Ranges specify a starting letter or number and a finishing letter or number. Ranges are a shorthand way of stating you want all the values in between the start position and the end position.

```ruby
# A range of numbers from 1 to 9
1..9
# A range of characters from 'a' to 'z'
'a'..'z'
# A range of characters from 'A' to 'Z'
'A'..'Z'
```

The first three examples are probably familiar to you. When you are writing out a long list of known things we often times abbreviate the list with a series of dots (called an Ellipsis and is usually stated with three dots ‘…’). In ruby this abbreviation is done with two dots.

IRB

```
2.1.1 :001> 1..9
```

```
            => 1..9
2.1.1 :002> (1..9).to_a
            => [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

A range is similar to an Array and can stand in for an array if you convert the range to an array with the `to_a` method. The first example shows the range we described. In the second example we convert the range into an array. This will show us every value in between.

Two of our ranges represented characters in the alphabet. We wanted a range of all the lower case letters and a separate range of all the upper case letters. We can also define a range which is both the upper case and the lower case letters.

IRB

```
2.1.1 :001> 'A'..'z'
            => "A".."z"
2.1.1 :002> ('A'..'z').to_a
            ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "
            L", "M", "N", "O", "P", "Q", "R", "S", "T", "U", "V", "W"
            , "X", "Y", "Z", "[", "\\", "]", "^", "_", "`", "a", "b",
             "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "
            n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y"
            , "z"]
```

We define a range from an upper case 'A' to a lower case 'z'. That seemed to work correctly. When we converted to an array it seems to have included a bunch of extra characters. Why did that happen?

All of the characters you can possibly type are stored somewhere in a big, long list. What we are seeing above is a subset of that big list. Someone decided awhile ago to store a few extra characters between the upper case letters and the lower case letters.

We can see an even bigger list if we create a range between the space character ' ' and a lower case 'z'.

IRB

```
2.1.1 :001> ' '..'z'
            => " ".."z"
2.1.1 :002> (' '..'z').to_a
            [" ", "!", "\"", "#", "$", "%", "&", "'", "(", ")", "*",
            "+", ",", "-", ".", "/", "0", "1", "2", "3", "4", "5", "6
            ", "7", "8", "9", ":", ";", "<", "=", ">", "?", "@", "A",
             "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L", "
            M", "N", "O", "P", "Q", "R", "S", "T", "U", "V", "W", "X"
            , "Y", "Z", "[", "\\", "]", "^", "_", "`", "a", "b", "c",
             "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "
o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z"]
```

Wow. That is a long list of characters. The best part about the list is that it contains nearly all the possible characters we could write in a message. This is something that we can use to our advantage when creating our cipher. This will save us many keystrokes by using Ranges that we convert to Arrays.

Exercise

Create an array from the range 'A'..'Z'

Create an array from the range 'a'..'z'

Create an array from the range '0'..'9'

Create an array from the range ' '..'z'

Array Rotate

Array has a special method called `rotate` which accepts a number of places to rotate the entire list. This is exactly what we did manually when we wrote out all the letters on a piece of paper.

IRB

```
2.1.1 :001> ('a'..'z').to_a
            => ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k"
            , "l", "m", "n", "o", "p", "q", "r", "s", "t", "u", "v",
2.1.1 :002> "w", "x", "y", "z"]
            ('a'..'z').to_a.rotate(13)
            => ["n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x"
            , "y", "z", "a", "b", "c", "d", "e", "f", "g", "h", "i",
            "j", "k", "l", "m"]
```

```
["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m",
"n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z"]
["n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z",
"a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m"]
```

Array's `rotate` method will allow you to easily create a cipher with any possible rotation.

Exercise

Create an array from the range ‘A’..’Z’

Create an array from the range ‘A’..’Z’ that is rotated by 1

Create an array from the range ‘0’..’9’ that is rotated by 5

Creating our cipher

1. Define the amount to rotate
2. Create an array of our list of characters.
3. Create a second array that is a list of characters rotated by the amount to rotate.
4. Create a Hash with the first list as the keys and the second list as the values.

IRB

```
2.1.1 :001>  rotation = 13
             => 13
2.1.1 :002>  characters = ('a'..'z').to_a
             => ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k"
             , "l", "m", "n", "o", "p", "q", "r", "s", "t", "u", "v",
2.1.1 :003>  "w", "x", "y", "z"]
             rotated_characters = characters.rotate(rotation)
             => ["n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x"
2.1.1 :004>  , "y", "z", "a", "b", "c", "d", "e", "f", "g", "h", "i",
             "j", "k", "l", "m"]
             pairs = characters.zip(rotated_characters)
             => [["a", "n"], ["b", "o"], ["c", "p"], ["d", "q"], ["e",
              "r"], ["f", "s"], ["g", "t"], ["h", "u"], ["i", "v"], ["
2.1.1 :005>  j", "w"], ["k", "x"], ["l", "y"], ["m", "z"], ["n", "a"],
              ["o", "b"], ["p", "c"], ["q", "d"], ["r", "e"], ["s", "f
             "], ["t", "g"], ["u", "h"], ["v", "i"], ["w", "j"], ["x",
              "k"], ["y", "l"], ["z", "m"]]
             Hash[pairs]
             {"a"=>"n", "b"=>"o", "c"=>"p", "d"=>"q", "e"=>"r", "f"=>"
             s", "g"=>"t", "h"=>"u", "i"=>"v", "j"=>"w", "k"=>"x", "l"
             =>"y", "m"=>"z", "n"=>"a", "o"=>"b", "p"=>"c", "q"=>"d",
             "r"=>"e", "s"=>"f", "t"=>"g", "u"=>"h", "v"=>"i", "w"=>"j
             ", "x"=>"k", "y"=>"l", "z"=>"m"}
```

Everything up to the last two step were concepts that we have introduced.

We defined our amount of rotation as 13. We created an array of characters from the range ‘a’ to ‘z’. We created a new array of characters rotated by 13.

We combine the two lists together with a special Array method called `zip`. Zip works like a clothing zipper by merging the two lists together like the teeth of a zipper. One after the other.

```
characters = ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k
", "l", "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x"
, "y", "z"]
rotated_characters = ["n", "o", "p", "q", "r", "s", "t", "u", "v",
 "w", "x", "y", "z", "a", "b", "c", "d", "e", "f", "g", "h", "i",
"j", "k", "l", "m"]
characters.zip(rotated_characters)
  # [["a", "n"], ["b", "o"], ["c", "p"], ["d", "q"], ["e", "r"], [
"f", "s"], ["g", "t"], ["h", "u"],
  #  ["i", "v"], ["j", "w"], ["k", "x"], ["l", "y"], ["m", "z"], [
"n", "a"], ["o", "b"], ["p", "c"],
  #  ["q", "d"], ["r", "e"], ["s", "f"], ["t", "g"], ["u", "h"], [
"v", "i"], ["w", "j"], ["x", "k"],
  #  ["y", "l"], ["z", "m"]]
```

The reason that we zipped the two arrays together is because Hash has a special creation syntax where if you provide with pairs it will create keys and values from it. The four lines of code that you just wrote is equivalent to all the time and energy you spent building your first cipher.

From this point forward we will use the character range `' '..'z'`. This will give us all upper case letters, numbers, lower case letters, and a bunch of common punctuation symbols.

```
rotation = 13
characters = (' '..'z').to_a
rotated_characters = characters.rotate(rotation)
Hash[characters.zip(rotated_characters)]
```

Exercise

Create a new hash with the range ‘A’..’Z’

Create a new hash with the rotation 25

Create a new hash with the range ’ ‘..’z’ with a rotation of 14

Creating a new cipher method

We now want to update our cipher method to take the amount of rotation as a parameter.

```
def cipher(rotation)
  characters = (' '..'z').to_a
  rotated_characters = characters.rotate(rotation)
  Hash[characters.zip(rotated_characters)]
end
```

When we do this we have changed the *signature* of our cipher method. In programming we say that methods have a *signature*.

Previously the signature of `cipher` had no parameters. It looked like the following:

```
def cipher
```

We have now changed the *signature* to have a single parameter. It looks like the following:

```
def cipher(rotation)
```

We now need to update all the places we previously used `cipher` to use `cipher(value)`. We will need to update `encrypt_letter`.

Updating our `encrypt_letter` method

Let us revisit our `encrypt_letter` method:

```
def encrypt_letter(letter)
  lowercase_letter = letter.downcase
  cipher[lowercase_letter]
end
```

Our `encrypt_letter` method no longer needs to `downcase` our character because our updated `cipher` method supports upper case characters. We also need to change how the `cipher` method is called as it now requires a parameter (the rotation amount).

```
def encrypt_letter(letter)
  rotation = 13
  cipher_for_rotation = cipher(rotation)
  cipher_for_rotation[letter]
end
```

With that change it is a good time to check to ensure that everything still works.

IRB

```
2.1.1 :001> load './encryptor.rb'
            => true
2.1.1 :002> e = Encryptor.new
            => #<Encryptor:0x007f7f391613f8>
2.1.1 :003> e.encrypt_letter("a")
            => "n"
```

Everything should be exactly the same. That is great to know. However, we now want to make sure that we can change the rotation every time we encrypt a character. So we are going to need to update the signature of the `encrypt_letter` method to accept a new parameter. The amount of rotation.

```
def encrypt_letter(letter,rotation)
  cipher_for_rotation = cipher(rotation)
  cipher_for_rotation[letter]
end
```

Now when we attempt to encrypt a letter we are going to need to specify the rotation.

IRB

```
2.1.1 :001> load './encryptor.rb'
            => true
2.1.1 :002> e = Encryptor.new
            => #<Encryptor:0x007f7f391613f8>
2.1.1 :003> e.encrypt_letter("a",13)
            => "n"
2.1.1 :004> e.encrypt_letter("a",1)
            => "b"
```

Updating our `encrypt` method

Now that the `encrypt_letter` expects two arguments, we need to rework `encrypt` to send it two arguments.

Currently the signature of `encrypt` looks like this:

```
def encrypt(string)
```

Change the method so it:

1. takes a second argument named `rotation`
2. passes that `rotation` into the call to `encrypt_letter`

Testing

When I test my method, here's the output:

IRB

```
2.1.1 :001> e.encrypt("Hello", 13)
            => "Uryy!"
2.1.1 :002> e.encrypt("Hello World", 13)
            => "Uryy!-d!$yq"
```

Wow. That is great. No one will know what we are saying to each other!

Writing `decrypt`

Speaking of which, we need to rework our `decrypt` method.

Depending on how you wrote the method originally, this might be easy or it might be hard. Consider this:

Decrypting is the opposite of encrypting. In our current process encrypting means moving forward `rotation` number of spots in the character map. Decrypting is then moving backwards the same number of spots.

Implement your own version of `decrypt` that can successfully match these results:

IRB

```
2.1.1 :001> load './encryptor.rb'
            => true
2.1.1 :002> e = Encryptor.new
            => #<Encryptor:0x00000108090b10>
2.1.1 :003> encrypted = e.encrypt("Hello, World!", 10)
            => "Rovvy6*ay!vn+"
2.1.1 :004> e.decrypt(encrypted, 10)
            => "Hello, World!"
2.1.1 :005> encrypted = e.encrypt("Hello, World!", 16)
            => "Xu!!$<0g$'!t1"
2.1.1 :006> e.decrypt(encrypted, 16)
            => "Hello, World!"
```

Now our encryption engine can flexibly use any rotation number!

Working with Files

Your encryption engine is cool for encrypting a few words, but what about a whole file? Using what we've already built, it's not too hard.

Experimenting with File I/O

Let’s first play with "File I/O" in IRB. When we say "I/O" we mean "Input / Output".

We’ll load a plain message in as input, then encrypt it, and output a new file with the encrypted message. We could then transmit that encrypted file, maybe as an email attachment, then our trusted correspondent can decrypt it back to a plain file.

File I/O in Ruby is much easier than many other programming languages. Let’s do I/O backwards and output a file first.

File Handles

Whenever we work with files we create a *file handle*. You can think of this as a connection between the program and the file system which holds the files.

It wouldn’t be accurate to say that a program holds or contains a file. Instead, we have this connection to the file system and can ask that connection to read in data from the file or write data out to it.

Outputting to a File

Let’s start by outputting some text to a file. Try this in IRB:

IRB

```
2.1.1 :001> out = File.open("sample.txt", "w")
2.1.1 :002> out.write("Hello, World!")
2.1.1 :003> out.write("This is a file, hooray.")
2.1.1 :004> out.close
```

When you run that then change back to SublimeText, you may see the `sample.txt` file pop up on the left side. If not, go to the FILE menu, click OPEN, and find `sample.txt`

What do you notice about this file? There’s something that isn’t quite right about how it writes out the text – the two lines of text are on the same line of the output file.

Try the instructions above, but add a `\n` to the end of the strings that you write out. This is a special marker to create a "new line".

Reading a File

The first line of the previous example was this:

```ruby
out = File.open("sample.txt", "w")
```

See that `"w"`? What was that for? When we create a file handle, a connection to the file system, we have to tell Ruby what kind of connection we want. Are we going to *w*rite to the file (`"w"`), *r*ead from it (`"r"`), or both (`"rw"`)?

Previously we wanted to write to the file, so we used the mode `"w"`. Now we want to read from the file, so we'll use `"r"` like this:

```
input = File.open("sample.txt", "r")
input.read
```

You'll see that the content of the file has been read back in, but it looks weird with the `"\n"` newlines in there. To see it printed with the lines broken apart, try this:

```
input = File.open("sample.txt", "r")
puts input.read
```

File Cursor

Now for something strange. Assuming you just did the previous example, try running just this instruction again:

```
puts input.read
```

What do you get out? Is that what you expected?

Probably not. When you open a file for reading you start with a "cursor".

Imagine you had a piece of paper with words on it. When you first look at the paper, you could put your finger on the first word on the page. This is your cursor.

If someone told you to read the page, you'd move the cursor along word by word, line by line, until you got to the end. When the cursor was on the last word, you'd stop.

File handles work the same way. When we first opened the file the cursor was on the first letter of the file. When we said `.read` it read back all the contents of the file.

But then the cursor was at the end of the file. If we call `.read` again we'll just get back `nil` because there's nothing left in the file.

If you wanted to read the file from the beginning again, you could do this:

```
input.rewind
puts input.read
```

Creating a Secret Message

Now we need a message to encrypt.

Using SublimeText, create a new file by going to the FILE menu and clicking NEW FILE.

In this file, create a short secret message that is at least three lines long.

Once you have the content, save it with the name `secret.txt` in the same directory as your `encryptor.rb` program.

Writing an `encrypt_file` Method

Let's start a new method in `encryptor.rb` like this:

```ruby
def encrypt_file(filename, rotation)

end
```

This method will take in two parameters, the filename of the file to be encrypted and the number of letters to rotate.

Pseudocode

Add this pseudocode into the method as comments:

1. Create the file handle to the input file
2. Read the text of the input file
3. Encrypt the text
4. Create a name for the output file
5. Create an output file handle
6. Write out the text
7. Close the file

Implement It

You've seen all the components that you need here. Figure out how to implement this method on your own. Here are a few notes to help you:

1. Use the filename variable from the parameter with the `File.open` call. Remember to specify the right read/write mode.
2. Just call the same method you did before to read the contents. You'll need to save this into a variable.
3. Call your `.encrypt` method passing in the text from step 2 and the rotation parameter
4. Name the output file the same as the input file, but with `".encrypted"` on the end. So an input file named `"sample.txt"` would generate a file named `"sample.txt.encrypted"`. Store the name in a variable.
5. Create a new file handle with the name from step 4 and remember the correct read/write mode.
6. Use the `.write` method like before.
7. Call `.close` on the output file handle

Test It

Run your code from IRB:

IRB

```
2.1.1 :001> load './encryptor.rb'
            => true
2.1.1 :002> e = Encryptor.new
            => #<Encryptor:0x007f7f3916be98>
2.1.1 :003> e.encrypt_file("sample.txt", 5)
            => nil
```

You get back `nil` because the `.close` method you called on the output file returns `nil`.

Open the `sample.txt.encrypted` in SublimeText. Does it look like a bunch of junk? Hopefully so! No one is going to be able to read your secret message.

Writing a `decrypt_file` Method

But did it really work? We can't know until we write a `decrypt_file` method.

Method Signature

The method should look like this:

```
def decrypt_file(filename, rotation)

end
```

Pseudocode

The pseudocode is almost the same:

1. Create the file handle to the encrypted file
2. Read the encrypted text
3. Decrypt the text by passing in the text and rotation
4. Create a name for the decrypted file
5. Create an output file handle
6. Write out the text
7. Close the file

You know how to do most of this. Here are two tricky parts:

Step 1. Read File Handle

For the very first step, where you create the file handle, Ruby will fail to detect which language the file is written in because of all the strange characters. You need to put a little more information in the read/write mode declaration like this:

```ruby
input = File.open(filename, "r")
```

Step 4. Output Filename

For the output filename, it'd be nice if we could call it something like `"sample.txt.decrypted"`. You could create that string using the `.gsub` method like this:

```ruby
output_filename = filename.gsub("encrypted", "decrypted")
```

Other than that, you're on your own!

Testing the Whole Process

Let's see the whole thing work together:

IRB

```
2.1.1 :001> load './encryptor.rb'
2.1.1 :002> e = Encryptor.new
2.1.1 :003> e.encrypt_file("sample.txt", 11)
2.1.1 :004> e.decrypt_file("sample.txt.encrypted", 11)
```

Then open `"sample.txt.decrypted"` and see how it looks.

If it matches your input file, then your encryption engine is complete!

Cracking Encryption

Sending encrypted messages to your friends has made others envious. Other people have started to encrypt the messages they send to each other. You intercept one of these messages.

```
"f w)0/6X0// -6C6` ''46j$( "
```

You know that the message is using a rotation encryption scheme (the person that sent it finished the same tutorial as you). However, what you do not know is the rotational number. What rotation number are they using?

Finding which rotation

To understand the encrypted message you need to figure out the rotation number they used. Knowing that number will allow you to change your decryption tools to get the original message. How do you find the rotation?

Ask the writer or receiver of the message to tell you what rotational number they are using.

Ask the writer or receiver of the message to tell you what rotational number they are using. Decrypt the message and look at the output and see if the message looks correct.

The solution involves very little programming. It instead relies on your ability to get people to give you information. Surprisingly people will volunteer this information. Especially if you are able to convince them you are on their team. Of course, the person telling you the rotation value may not be telling the truth.

Guess a rotational number based on something you may know about the writer or receiver of the message.

Guess a rotational number based on something you may know about the writer or receiver of the message. Decrypt the message and Look at the output and see if the message looks correct.

This solution involves you trying to understand what number a person might choose. Does the writer of this use the same rotational value when sending you encrypted messages? Does the writer or receiver have a favorite number? Finding the solution requires you to make a guess, change your decryption code, run it, and then review the message.

Like a game of hangman, the number of possible choices grows smaller with each choice. However, making several wrong guesses can be time consuming.

Decrypt the message using every rotational number. Looking at all the output and see which message looks correct.

Decrypt the message using every rotational number. Looking at all the output and see which message looks correct.

This solution is the one that we can best solve with code. Our current decryption method allows us to specify a single rotational number. We need to create a new method that will generate all possible outputs for all possible rotational numbers.

Step 1. Solving this problem using decrypt

We can solve this problem by using our existing `decrypt` method. We can call it for every possible rotational number. Looking at the results each time.

IRB

```
2.1.1 :001>  load './encryptor.rb'
2.1.1 :002>  e = Encryptor.new
2.1.1 :003>  e.decrypt('f w)0/6X0// –6C6` ''46j$( ',1)
             => 'ezv(/.5W/..z,5B5_z&&35i#'z'
2.1.1 :004>  e.decrypt('f w)0/6X0// –6C6` ''46j$( ',2)
             => 'dyu'.–4V.––y+4A4^y%%24h\"&y'
2.1.1 :005>  e.decrypt('f w)0/6X0// –6C6` ''46j$( ',3)
```

```
             => 'cxt&-,3U-,,x*3@3]x$$13g!%x'
2.1.1 :006> e.decrypt('f w)0/6X0// -6C6` ''46j$( ',4)
             => 'bws%,+2T,++w)2?2\\w##02f $w'
```

Trying to crack the encrypted this way is very tedious. We would need to keep doing this until we found the right one. This could result in a lot of attempts. More importantly, if we wanted to crack another message in the future we would have to do this again. This is another situation where we can use looping to simplify our job.

IRB

```
2.1.1 :001> load './encryptor.rb'
2.1.1 :002> e = Encryptor.new
2.1.1 :003> (' '..'z').to_a.size.times do |attempt|
2.1.1 :004> puts e.decrypt('ENCRYPTED',attempt)
2.1.1 :005> end
```

To figure out all the possible combinations we need to consider all the possible characters. That is why we needed to use the same range of characters again and figure out how many of them we support. The decrypted message should appear in a list alongside 90 other garbled messages. Take your time to find the message.

Congratulations you have cracked the code!

Step 2: Define a method named `crack` which accepts our encrypted message.

We cracked the code. You have intercepted a new message. It is time to crack this one.

```
"\\qmz&%,N&%%q#,9,Vqxx*,`uyq"
```

We can solve this problem again using the code we wrote above:

```
(' '..'z').to_a.count.times do |attempt|
  puts e.decrypt('ENCRYPTED',attempt)
end
```

However, writing that out every single time would be tedious and time-consuming. We should instead make it a standard part of our Encryptor class. That way we can call it again when we have new messages in the future to crack.

Let's add a new `crack` method to our Encryptor. The `crack` method should accept an encrypted message. However, we want to change it slightly. Instead of outputting the messages immediately with the `puts` method we want to collect them all and send return them. This will allow us to save them to a file if needed.

```ruby
class Encryptor
  # ... other Encryptor methods ...

  def supported_characters
    (' '..'z').to_a
  end

  def crack(message)
    supported_characters.count.times.collect do |attempt|
      decrypt(message,attempt)
    end
  end
end
```

Now let's try our new `crack` method:

IRB

```
2.1.1 :001> load './encryptor.rb'
2.1.1 :002> e = Encryptor.new
2.1.1 :003> e.crack "ENCRYPTED MESSAGE 2"
```

Congratulations. You now have a way to thwart your enemies and spy on your friends. Most importantly, it should show you that using this form of encryption (ROT-#) is not safe for very long.

Further Exercise

Real-Time Encryption

You want to start using your encryption in more of your communication. Writing your original message to a file, encrypting it, and opening the file requires a lot of effort. It is not well suited for small amounts of text like a chat message or text messages.

- Create a system that will allow you to type a unencrypted message and have the encrypted version appear
- Create a system that will allow you to type an encrypted message and have the unencrypted message appear

Password Protecting Your Encryptor

The encryptor program does a fair job at protecting your correspondence. The messages you send to and from your friends are safe from prying eyes. However, your security would be compromised if your encryptor code fell into the wrong hands.

- Add a simple password prompt when running encryptor

- Protect your simple password by using your encryption
- Use Ruby's MD5 Hash of the password and store that in the file
- Use Ruby's MD5 Hash to compare incoming password attempts to see if they match

Building Better a ROT

We saw how easy it was to break the encryption when we used a single rotation value. We could build better encryption if we used multiple rotations within the same document.

- Pick three different numbers

Select three numbers that you can remember. These three numbers will be the three encryption rotations that we will cycle through as we encrypt each letter.

- When encrypting each character, continue to cycle through the three numbers you selected as your encryption ROT value.

Encrypting each letter with a different rotation will make it hard for for someone to crack your messages. Even if they were able to figure out that you were using three different rotations they would still need to generate all possible outputs to see which one looked correct.

Assuming you are rotating through the same 91 characters, choosing three numbers would require a cracker to look through **753571** possible combinations to figure out what you wrote. Each number you add would make that amount increase even more!

Dpohsbuvmbujpot\"