Pointless: Learning by Example

Let's see an example to get a feel for the language. I'll show you how I wrote the following program, which can encode and decode a message using a simple cipher.

```
alphabet = chars("abcdefghijklmnopqrstuvwxyz")

-- Shift a single letter 13 places

fn shift(letter)
  index = list.indexOf(alphabet, str.toLower(letter))

  if index == none then
    letter
  else
    shifted = alphabet[(index + 13) % 26]

    if str.isUpper(letter) then
      str.toUpper(shifted)
    else
      shifted
    end
  end
end

-- Encode or decode a message with the ROT13 cipher
-- https://en.wikipedia.org/wiki/ROT13

fn cipher(message)
  message
    | chars
    $ shift
    | join("")
end

-- What does this print?

"Uryyb jbeyq!"
  | cipher
  | print
```

The code above implements the *ROT13 cipher*: a special case of the *Caesar cipher* where each letter in a message is replaced with the 13th letter after it in the English alphabet, wrapping around to the start if needed.

```
before encoding: 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
after encoding:  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
```

For example, the ROT13 encoding of the word `"cat"` is `"png"`.

Shifting Letters

To implement the cipher, we start by defining the variable `alphabet` as a list containing the `26` letters of the English alphabet. We do this using the `chars(string)` function to split a string of these letters into a list.

```
alphabet = chars("abcdefghijklmnopqrstuvwxyz")
```

```
alphabet
["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"]
```

Displaying Results

In this tutorial, the values produced by each piece of code are displayed automatically. In a normal Pointless program you would use the `print` function to display these results, for example `print(alphabet)`.

We can use the `list.indexOf(list, value)` function to get the index of a character within `alphabet`. For example, the letter `"i"` (the 9th letter in the alphabet) will have index `8` (Pointless lists are *0-indexed*).

```
list.indexOf(alphabet, "i")
```

```
8
```

In addition to finding the index of a letter, we can also do the reverse: get a letter from the alphabet based on its index.

```
alphabet[8]
```

```
"i"
```

At its core, our cipher will do the following for each letter:

- Convert the letter to an index.
- Shift the index value, wrapping back to the start of the alphabet if necessary.
- Convert the shifted index back to a letter.

We can write a few lines of code that do just that.

```
letter = "i"
index = list.indexOf(alphabet, letter)
shifted = alphabet[(index + 13) % 26]
```

```
shifted
"v"
```

Here, we're using `(index + 13) % 26` as our shifted index value. When `index + 13` is greater than or equal to `26`, the modulus operator will wrap the number back around to get a value between `0` and `25`. If we didn't

use the modulus operator here, our code wouldn't work correctly for letters in the second half of the alphabet.

Let's put this code into a new function called `shift`. The function will take a single letter and return the corresponding ROT13 encoded letter. We can use the `fn` keyword to define a new function.

```
fn shift(letter)
  index = list.indexOf(alphabet, letter)
  shifted = alphabet[(index + 13) % 26]
  shifted
end
```

> **Implicit Return**
>
> Pointless has a `return` keyword that can be used to return values from functions; however, as in other languages like Rust and Ruby, functions in Pointless return the value of their final expression by default (in this case the variable `shifted`), so the `return` is usually omitted.

We can call `shift` to make sure it's working properly.

```
shift("c")
shift("a")
shift("t")
```

```
"p"
"n"
"g"
```

Shifting Whole Strings

Ultimately we want to be able to encode an entire string at once, instead of character-by-character. We'll define a new function `cipher` that takes a string `message`, calls the `shift` function for each letter in `message`, joins the shifted letters together, and returns the resulting encoded string.

```
fn cipher(message)
  message
    | chars
    $ shift
    | join("")
end
```

Like before, we're using the `chars(string)` function to transform a string (`message`) into a list of characters; however, this time we're calling it using "pipeline" syntax. In this syntax, the pipe `|` operator calls the function that comes after it with the argument value that comes before it.

```
"cat" | chars
```

```
["c", "a", "t"]
```

In Pointless, the following two forms are equivalent.

```
chars(message)
message | chars
```

There's a second form of the pipeline syntax which uses the map `$` operator. Like the pipe `|` operator, the map `$` operator calls the function that comes after it with the argument value that comes before it, with a twist:

- It requires that its argument value is a list (or another iterable type).
- It calls the function on **each** element in the argument list.
- It returns the result of each of these calls as a new list.

```
"cat" | chars $ shift
```

```
["p", "n", "g"]
```

As a final step, the `cipher` function takes the list of shifted letters and passes them to the `join(values, sep)` function, which joins them together into a single string with the separator `sep` in between. We don't want anything between the letters, so we'll use an empty separator `""`.

```
"cat" | chars $ shift | join("")
```

```
"png"
```

Unlike `chars` and `shift`, `join` takes two arguments, `values` and `sep`. The pipe `|` operator will supply the first argument (`values`), but we need to specify a value for `sep` as well.

In Pointless, the following two forms are equivalent.

```
join(shifted, "")
shifted | join("")
```

Why Use Pipeline Syntax?

We didn't have to use pipeline operators to write `cipher`. We could have written it like this instead.

```
fn cipher(message)
  join(list.map(chars(message), shift), "")
end
```

However, I often prefer using the pipeline syntax over nested function calls; it lets us structure our code as a sequence of transformations, which can make it easier to understand and modify.

Additionally, when writing an expression with more than one pipeline transformation, I like to break the code up into multiple lines. So, instead of writing this.

```
"cat" | chars $ shift | join("")
```

I write this.

```
"cat"
  | chars
  $ shift
  | join("")
```

Encoding and Decoding

Now let's call `cipher` to make sure it's working properly.

```
cipher("cat")
cipher("png")
```

```
"png"
"cat"
```

This example shows us something interesting: if we pass an encoded string (like `"png"`) to `cipher`, we get back the original unencoded string! This happens because the ROT13 cipher uses `13` as the shift value. When we call `cipher` on an encoded string, the encoded letters (which were already shifted `13` places from the original letters) are shifted `13` places again, bringing the total shift amount to `26`. Since our alphabet is `26` letters long, this process wraps each letter back to its original value. **This means that we can use `cipher` for both encoding and decoding!**

We can verify this by calling `cipher` twice on the same message and checking that we get the original message back.

```
cipher(cipher("cat"))
```

```
"cat"
```

Uppercase Letters

Currently, our code looks like this.

```
alphabet = chars("abcdefghijklmnopqrstuvwxyz")

fn shift(letter)
  index = list.indexOf(alphabet, letter)
  shifted = alphabet[(index + 13) % 26]
  shifted
end

fn cipher(message)
  message
    | chars
    $ shift
    | join("")
end
```

We still need to make a couple of changes. The first issue is uppercase letters: the `list.indexOf` function that we use in `shift` is case-sensitive, so our code won't work for messages containing uppercase letters. We can solve this by having `shift` convert `letter` to lowercase before finding its index.

```
fn shift(letter)
  index = list.indexOf(alphabet, str.toLower(letter))
  shifted = alphabet[(index + 13) % 26]
  shifted
end
```

This change allows `cipher` to handle uppercase letters, but we end up losing information about which letters were capitalized in the original message.

```
cipher("Cat")
cipher(cipher("Cat"))
```

```
"png"
"cat"
```

We'll update the definition for our cipher so that uppercase letters in `message` get translated to uppercase letters in the final encoded string.

```
before encoding: 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
after encoded:   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

before encoding: 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
after encoded:   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
```

We can add this behavior to `shift` using a conditional expression. This conditional uses the `str.isUpper` function to check whether `letter` is uppercase and convert `shifted` to uppercase when necessary.

```
fn shift(letter)
  index = list.indexOf(alphabet, str.toLower(letter))
  shifted = alphabet[(index + 13) % 26]

  if str.isUpper(letter) then
    str.toUpper(shifted)
  else
    shifted
  end
end
```

Our `cipher` function now preserves letter case.

```
cipher("Cat")
cipher(cipher("Cat"))
```

```
"Png"
"Cat"
```

Conditionals Are Expressions

In Pointless, as in many other functional languages, **conditionals are expressions**, which means we can use them within larger pieces of code such as variable assignments and return expressions.

```
a = 7
b = 8
maximum = if a > b then a else b end
```

```
maximum
```

```
8
```

Non-Alphabetic Characters

The final piece to consider is non-alphabetic characters. Currently, if we call `shift` with a non-alphabetic character (like `"!"`), the call to `list.indexOf` will return `none`. This will cause an error later when we try to do math with the `none` value as though it were a number. We can fix this by having `shift` check whether `list.indexOf` returned `none` and returning `letter` unmodified if it did.

```
fn shift(letter)
  index = list.indexOf(alphabet, str.toLower(letter))

  if index == none then
    letter
  else
    shifted = alphabet[(index + 13) % 26]

    if str.isUpper(letter) then
      str.toUpper(shifted)
    else
      shifted
    end
  end
end
```

Conditionals with Multiple Statements

As this new code demonstrates, the `then` and `else` branches of a conditional can contain multiple statements. As with functions, a conditional expression takes on the value of the *final expression* of its matching branch.

Now `shift` and `cipher` will translate alphabetic characters and pass non-alphabetic characters through unmodified.

```
cipher("Cat!")
cipher(cipher("Cat!"))
```

```
"Png!"
"Cat!"
```

Our code is now complete!

Finishing Up

So, what's the secret message?

```
cipher("Uryyb jbeyq!")
```

```
"Hello world!"
```