Case Study: Roman Numerals

WE'LL COVER THE FOLLOWING

You've most likely seen Roman numerals, even if you didn't recognize them. You may have seen them in copyrights of old movies and television shows ("Copyright `MCMXLVI`" instead of "Copyright `1946`"), or on the dedication walls of libraries or universities ("established `MDCCCLXXXVIII`" instead of "established `1888`"). You may also have seen them in outlines and bibliographical references. It's a system of representing numbers that really does date back to the ancient Roman empire (hence the name).

In Roman numerals, there are seven characters that are repeated and combined in various ways to represent numbers.

- I = 1
- V = 5
- X = 10
- L = 50
- C = 100
- D = 500
- M = 1000

The following are some general rules for constructing Roman numerals:

- Sometimes characters are additive. `I` is `1`, `II` is `2`, and `III` is `3`. `VI` is `6` (literally, "`5` and `1`"), `VII` is `7`, and `VIII` is `8`.
- The tens characters (`I`, `X`, `C`, and `M`) can be repeated up to three times. At `4`, you need to subtract from the next highest fives character. You

At `4`, you need to subtract from the next highest fives character. You can't represent `4` as `IIII`; instead, it is represented as `IV` ("1 less than 5"). `40` is written as `XL` ("`10` less than `50`"), `41` as `XLI`, `42` as `XLII`, `43` as `XLIII`, and then `44` as `XLIV` ("`10` less than `50`, then `1` less than `5`").

- Sometimes characters are... the opposite of additive. By putting certain characters before others, you subtract from the final value. For example, at `9`, you need to subtract from the next highest tens character: `8` is `VIII`, but `9` is `IX` ("`1` less than `10`"), not `VIIII` (since the `I` character can not be repeated four times). `90` is `XC`, `900` is `CM`.
- The fives characters can not be repeated. `10` is always represented as `X`, never as `VV`. `100` is always `C`, never `LL`.
- Roman numerals are read left to right, so the order of characters matters very much. `DC` is `600`; `CD` is a completely different number (`400`, "`100` less than `500`"). `CI` is `101`; `IC` is not even a valid Roman numeral (because you can't subtract `1` directly from `100`; you would need to write it as `XCIX`, "`10` less than `100`, then `1` less than `10`").

Checking For Thousands

What would it take to validate that an arbitrary string is a valid Roman numeral? Let's take it one digit at a time. Since Roman numerals are always written highest to lowest, let's start with the highest: the thousands place. For numbers 1000 and higher, the thousands are represented by a series of M characters.

```
import re
pattern = '^M?M?M?$'                #①
print (re.search(pattern, 'M'))     #②
#<_sre.SRE_Match object at 0106FB58>

print (re.search(pattern, 'MM'))    #③
#<_sre.SRE_Match object at 0106C290>

print (re.search(pattern, 'MMM'))   #④
#<_sre.SRE_Match object at 0106AA38>

re.search(pattern, 'MMMM')          #⑤
print (re.search(pattern, ''))      #⑥
#<_sre.SRE_Match object at 0106F4A8>
```

① This pattern has three parts. `^` matches what follows only at the beginning of the string. If this were not specified, the pattern would match no matter where the `M` characters were, which is not what you want. You want to make sure that the `M` characters, if they're there, are at the beginning of the string. `M?` optionally matches a single `M` character. Since this is repeated three times, you're matching anywhere from zero to three `M` characters in a row. And `$` matches the end of the string. When combined with the `^` character at the beginning, this means that the pattern must match the entire string, with no other characters before or after the `M` characters.

② The essence of the `re` module is the `search()` function, that takes a regular expression (pattern) and a string (`'M'`) to try to match against the regular expression. If a match is found, `search()` returns an object which has various methods to describe the match; if no match is found, `search()` returns `None`, the Python null value. All you care about at the moment is whether the pattern matches, which you can tell by just looking at the return value of `search()`. `'M'` matches this regular expression, because the first optional `M` matches and the second and third optional M characters are ignored.

③ `'MM'` matches because the first and second optional M characters match and the third `M` is ignored.

④ `'MMM'` matches because all three `M` characters match.

⑤ `'MMMM'` does not match. All three `M` characters match, but then the regular expression insists on the string ending (because of the `$` character), and the string doesn't end yet (because of the fourth `M`). So `search()` returns `None`.

⑥ Interestingly, an empty string also matches this regular expression, since all the `M` characters are optional.

Checking For Hundreds

? makes a pattern optional.

The hundreds place is more difficult than the thousands, because there are several mutually exclusive ways it could be expressed, depending on its value.

- 100 = C
- 200 = CC

- 300 = CCC
- 400 = CD
- 500 = D
- 600 = DC
- 700 = DCC
- 800 = DCCC
- 900 = CM

So there are four possible patterns:

- `CM`
- `CD`
- Zero to three C characters (zero if the hundreds place is 0)
- `D`, followed by zero to three `C` characters

The last two patterns can be combined:

- an optional `D`, followed by zero to three `C` characters

This example shows how to validate the hundreds place of a Roman numeral.

```
import re
pattern = '^M?M?M?(CM|CD|D?C?C?C?)$'          #①
print (re.search(pattern, 'MCM') )            #②
#<_sre.SRE_Match object at 01070390>

print (re.search(pattern, 'MD') )             #③
#<_sre.SRE_Match object at 01073A50>

print (re.search(pattern, 'MMMCCC'))          #④
#<_sre.SRE_Match object at 010748A8>

re.search(pattern, 'MCMC')                    #⑤
print (re.search(pattern, ''))                #⑥
#<_sre.SRE_Match object at 01071D98>
```

① This pattern starts out the same as the previous one, checking for the beginning of the string (`^`), then the thousands place (`M?M?M?`). Then it has the new part, in parentheses, which defines a set of three mutually exclusive patterns, separated by vertical bars: `CM`, `CD`, and `D?C?C?C?` (which is an

optional `D` followed by zero to three optional `C` characters). The regular

expression parser checks for each of these patterns in order (from left to right), takes the first one that matches, and ignores the rest.

② `'MCM'` matches because the first `M` matches, the second and third `M` characters are ignored, and the `CM` matches (so the `CD` and `D?C?C?C?` patterns are never even considered). `MCM` is the Roman numeral representation of `1900`.

③ `'MD'` matches because the first `M` matches, the second and third `M` characters are ignored, and the `D?C?C?C?` pattern matches `D` (each of the three `C` characters are optional and are ignored). `MD` is the Roman numeral representation of `1500`.

④ `'MMMCCC'` matches because all three `M` characters match, and the `D?C?C?C?` pattern matches `CCC` (the `D` is optional and is ignored). `MMMCCC` is the Roman numeral representation of `3300`.

⑤ `'MCMC'` does not match. The first `M` matches, the second and third `M` characters are ignored, and the `CM` matches, but then the `$` does not match because you're not at the end of the string yet (you still have an unmatched `C` character). The `C` does not match as part of the `D?C?C?C?` pattern, because the mutually exclusive `CM` pattern has already matched.

⑥ Interestingly, an empty string still matches this pattern, because all the `M` characters are optional and ignored, and the empty string matches the `D?C?C?C?` pattern where all the characters are optional and ignored.

Whew! See how quickly regular expressions can get nasty? And you've only covered the thousands and hundreds places of Roman numerals. But if you followed all that, the tens and ones places are easy, because they're exactly the same pattern. But let's look at another way to express the pattern.