Qualified by andela Assessment Result

Tanyaradzwa Chisepo

Code2040: Programming Essentials [2023] Results

✉ tanyachisepo04@gmail.com

Assessment Summary

100%

1 h, 28 m, 18 s
Active Time

- Tanyaradzwa Chisepo **opened** this assessment on Friday, July 1, 2022 6:08 AM
- Tanyaradzwa Chisepo **started** this assessment on Friday, July 1, 2022 6:36 AM
- Tanyaradzwa Chisepo **submitted** this assessment after **1 day, 17 hours and 23 minutes** on Saturday, July 2, 2022 11:59 PM
- This student spent **1 hour and 28 minutes** active in the browser working on the assessment

Solutions Summary

Challenge	Score	Active Time
#1: RLE Decoder	100%	30 m, 31 s
#2: Adding Ordinal Indicator Suffixes	100%	38 m, 46 s
#3: Roman Numerals Decoder	100%	19 m, 1 s

Qualified by andela Assessment Code Solution

#1: RLE Decoder

Scoring

100%

7 / 7 Tests (1 Attempt)

Timing

30 m, 31 s Active Time

738 ms Run Time

Instructions

RLE Decoder

Run-length encoding (https://en.wikipedia.org/wiki/Run-length_encoding) is a form of lossless compression that works well on data characterized by repetition (for example, images with large blocks of the same color).

The run length decoder here works correctly on only very simple strings and could use improvement. It currently accepts a string such as `"4a4b3c1d1c"`, where each run of a repeated character is prefixed by a digit designating how many times the character should be repeated, and emits the uncompressed version `"aaaabbbbcccdc"`. However, it doesn't support input such as `"10a"` which should return `"aaaaaaaaaa"`.

Your task

Your task in this challenge is to add support for multiple digit prefixes in a compression string. Feel free to modify the provided code to the extent you wish.

Output will be limited to the lowercase alphabet, while input will be a valid run-length encoded sequence of alphanumeric characters or an empty string.

Examples

```
decodeRLE("6y");        ⟹  "yyyyyy"
decodeRLE("9q8i2o");    ⟹  "qqqqqqqqqiiiiiiiioo"
decodeRLE("10i14d");    ⟹  "iiiiiiiiiidddddddddddddd"
decodeRLE("5w6r1h3o");  ⟹  "wwwwwrrrrrrhooo"
```

```
decodeRLE("6y");        ⟹  "yyyyyy"
decodeRLE("9q8i2o");    ⟹  "qqqqqqqqqiiiiiiiioo"
decodeRLE("10i14d");    ⟹  "iiiiiiiiiidddddddddddddd"
decodeRLE("5w6r1h3o");  ⟹  "wwwwwrrrrrrhooo"
```

```
decode_rle("6y")        ⟹  "yyyyyy"
decode_rle("9q8i2o")    ⟹  "qqqqqqqqqiiiiiiiioo"
decode_rle("10i14d")    ⟹  "iiiiiiiiiidddddddddddddd"
decode_rle("5w6r1h3o")  ⟹  "wwwwwrrrrrrhooo"
```

```
decode("6y")        ⟹  "yyyyyy"
decode("9q8i2o")    ⟹  "qqqqqqqqqiiiiiiiioo"
decode("10i14d")    ⟹  "iiiiiiiiiidddddddddddddd"
decode("5w6r1h3o")  ⟹  "wwwwwrrrrrrhooo"
```

```
decode_rle("6y")        ⟹  "yyyyyy"
decode_rle("9q8i2o")    ⟹  "qqqqqqqqqiiiiiiiioo"
decode_rle("10i14d")    ⟹  "iiiiiiiiiidddddddddddddd"
decode_rle("5w6r1h3o")  ⟹  "wwwwwrrrrrrhooo"
```

```
decodeRLE("6y")         ⟹  "yyyyyy"
decodeRLE("9q8i2o")     ⟹  "qqqqqqqqqiiiiiiiioo"
decodeRLE("10i14d")     ⟹  "iiiiiiiiiidddddddddddddd"
decodeRLE("5w6r1h3o")   ⟹  "wwwwwrrrrrrhooo"
```

```
decodeRLE("6y");        ⟹  "yyyyyy"
decodeRLE("9q8i2o");    ⟹  "qqqqqqqqqiiiiiiiioo"
decodeRLE("10i14d");    ⟹  "iiiiiiiiiidddddddddddddd"
decodeRLE("5w6r1h3o");  ⟹  "wwwwwrrrrrrhooo"
```

```
decodeRLE("6y");        ⟹  "yyyyyy"
decodeRLE("9q8i2o");    ⟹  "qqqqqqqqqiiiiiiiioo"
decodeRLE("10i14d");    ⟹  "iiiiiiiiiidddddddddddddd"
decodeRLE("5w6r1h3o");  ⟹  "wwwwwrrrrrrhooo"
```

```
decodeRLE("6y");        ⟹  "yyyyyy"
decodeRLE("9q8i2o");    ⟹  "qqqqqqqqqiiiiiiiioo"
decodeRLE("10i14d");    ⟹  "iiiiiiiiiidddddddddddddd"
decodeRLE("5w6r1h3o");  ⟹  "wwwwwrrrrrrhooo"
```

```
RLEDecoder.decode("6y");        ⟹  "yyyyyy"
RLEDecoder.decode("9q8i2o");    ⟹  "qqqqqqqqqiiiiiiiioo"
RLEDecoder.decode("10i14d");    ⟹  "iiiiiiiiiidddddddddddddd"
RLEDecoder.decode("5w6r1h3o");  ⟹  "wwwwwrrrrrrhooo"
```

```
./decodeRLE "6y"        ⟹  "yyyyyy"
./decodeRLE "9q8i2o"    ⟹  "qqqqqqqqqiiiiiiiioo"
./decodeRLE "10i14d"    ⟹  "iiiiiiiiiidddddddddddddd"
./decodeRLE "5w6r1h3o"  ⟹  "wwwwwrrrrrrhooo"
```

```
RLEDecoder.Decode("6y");        ⟹  "yyyyyy"
RLEDecoder.Decode("9q8i2o");    ⟹  "qqqqqqqqqiiiiiiiioo"
RLEDecoder.Decode("10i14d");    ⟹  "iiiiiiiiiidddddddddddddd"
RLEDecoder.Decode("5w6r1h3o");  ⟹  "wwwwwrrrrrrhooo"
```

Solution Code

Python

```python
def decode_rle(encoded):
  # variable to store the decoded string
    decoded = ''
    #variable to store the digit prefix of every character
    number = ''
    for char in encoded:
      #checking to see if current character is a number or not
        if char.isdigit() :
```

```
            #allows for multiple digit numbers and stores the number
            number += char

        else:
          #appending the encoded string for the current character
            decoded += char * int(number)

            #resetting the digit prefix for the next characterr compression
            number = ''


    return decoded
```

Candidate's Tests

```
import unittest
from solution import decode_rle

class Test(unittest.TestCase):
    def test_single_digit_compression(self):
        self.assertEqual(decode_rle("4a4b3c1d1c"), "aaaabbbbcccdc")

    def test_double_digit_compression(self):
        self.assertEqual(decode_rle("10a"), "aaaaaaaaaa")

```

#2: Adding Ordinal Indicator Suffixes

Scoring

100%

4 / 4 Tests (8 Attempts)

Timing

38 m, 46 s Active Time

706 ms Run Time

Instructions

Task

Finish the function `numberToOrdinal`, which should take a number and return it as a string with the correct ordinal indicator suffix (in English). For example, `1` turns into `"1st"`.

For the purposes of this challenge, you may assume that the function will always be passed a non-negative integer. If the function is given `0` as an argument, it should return the string `"0"` without a suffix.

To help you get started, here is an excerpt from Wikipedia's page on Ordinal Indicators (https://en.wikipedia.org/wiki/Ordinal_indicator) :

> - *st* is used with numbers ending in 1 (e.g. 1st, pronounced first)
> - *nd* is used with numbers ending in 2 (e.g. 92nd, pronounced ninety-second)
> - *rd* is used with numbers ending in 3 (e.g. 33rd, pronounced thirty-third)
> - As an exception to the above rules, all the "teen" numbers ending with 11, 12 or 13 use -th (e.g. 11th, pronounced eleventh, 112th, pronounced one hundred [and] twelfth)
> - *th* is used for all other numbers (e.g. 9th, pronounced ninth).

Specification

```
{
"method": "number_to_ordinal",
"desc": "take a number and return it as a string with the correct English ordinal indicator
suffix",
"args": {
  "number": {"type": "Integer", "desc": "The number to be converted to a string ordinal"}
},
"returns": {"type": "String", "desc": "Returns a string ordinal based on the number."},
"constraints": ["`0 ≤ number ≤ 10000`"],
"examples": [
  {"args": [1], "returns": "1st"},
  {"args": [2], "returns": "2nd"},
  {"args": [3], "returns": "3rd"},
```

```
  {"args": [4], "returns": "4th"},
  {"args": [21], "returns": "21st"}
]
}
```

Solution Code

Python

```python
def number_to_ordinal(number):

    digit = number
    number = str(number)
    last_digit = int(number[-1])

    if digit >9:
        sec_last_digit = int(number[-2])

    if digit == 0:
      return "0"
    if number == '1' or  (last_digit == 1 and sec_last_digit != 1) :

        ordinal = number +'st';

    elif number == '2' or (last_digit == 2 and sec_last_digit != 1) :

        ordinal = number +'nd';
    elif number == '3' or (last_digit == 3 and sec_last_digit != 1) :

        ordinal = number +'rd';
    else:

        ordinal = number +'th';
    return ordinal
```

Candidate's Tests

```python
from unittest import TestCase
from solution import number_to_ordinal
TestCase.maxDiff = None

class TestSampleTests(TestCase):
    def test_should_handle_single_digits(self):
        """ should handle single digits """
        self.assertEqual(number_to_ordinal(1), "1st")
        self.assertEqual(number_to_ordinal(2), "2nd")
        self.assertEqual(number_to_ordinal(3), "3rd")
        self.assertEqual(number_to_ordinal(0), "0")
```

#3: Roman Numerals Decoder

Scoring

100%

3 / 3 Tests (1 Attempt)

Timing

19 m, 1 s Active Time

703 ms Run Time

Instructions

Task

Create a function that takes a Roman numeral as its argument and returns its value as a numeric decimal integer. You don't need to validate the form of the Roman numeral.

Modern Roman numerals are written by expressing each decimal digit of the number to be encoded separately, starting with the leftmost digit and skipping any `0`s. So `1990` is rendered `"MCMXC"` (`1000 = M`, `900 = CM`, `90 = XC`) and `2008` is rendered `"MMVIII"` (`2000 = MM`, `8 = VIII`). The Roman numeral for `1666`, `"MDCLXVI"`, uses each letter in descending order.

Example:

Input	Output
`"XXI"`	`21`
`"M"`	`1000`
`"MCMXC"`	`1990`
`"MMVIII"`	`2008`
`"MDCLXVI"`	`1666`

Specification

```
{
"method": "decode",
"desc": "Converts a Roman Numeral into a numeric decimal integer",
"args": {
  "roman": {"type": "String", "desc": "Roman Numberal to be converted"}
```

```
},
"returns": {"type": "Integer", "desc": "Evaluated decimal number"},
"examples": [
  {"args": ["XXI"], "returns": 21},
  {"args": ["MDCLXVI"], "returns": 1666},
  {"args": ["MMVIII"], "returns": 2008}
]
}
```

Solution Code

Python

```python
roman_dict = { 'M':1000, 'D': 500, 'C': 100, 'L': 50, 'X': 10,
    'I': 1, 'V': 5 }

def value(symbol):
    return (roman_dict[symbol[0]])


def decode(roman):


    res = 0
    sum = 0

    i = 0
    while i < (len(roman)):

        val_1 = value(roman[i])

        if (i + 1 < len(roman)):
            val_2 = value(roman[i+1])
            if val_1 >= val_2:

                sum  += val_1
                i += 1
            else:
                sum += val_2 - val_1
                i += 2
        else:
            sum += val_1
            i += 1


    return sum
```

Candidate's Tests

```python
import unittest
from solution import decode

class Test(unittest.TestCase):
    def test_should_work_for_individual_characters(self):
        self.assertEqual(decode("I"), 1)
    def test_should_handle_descending_values(self):
        self.assertEqual(decode("XXI"), 21)
    def test_should_handle_ascending_values(self):
        self.assertEqual(decode("IV"), 4)
```