Python Coding Interview

August 4, 2023

1 Interview Questions

1.1 Leap

Given a year, report if it is a leap year.

The tricky thing here is that a leap year in the Gregorian calendar occurs:

```
on every year that is evenly divisible by 4
  except every year that is evenly divisible by 100
    unless the year is also evenly divisible by 400
```

For example, 1997 is not a leap year, but 1996 is. 1900 is not a leap year, but 2000 is.

If your language provides a method in the standard library that does this look-up, pretend it doesn't exist and implement it yourself.

[2]:
```python
def is_leap(year:int):
    return year % 4 == 0 and year % 400 == 0
print(is_leap(1900))
```

```
False
```

1.2 Difference Of Squares

Find the difference between the square of the sum and the sum of the squares of the first N natural numbers.

The square of the sum of the first ten natural numbers is $(1 + 2 + ... + 10)^2 = 55^2 = 3025$.

The sum of the squares of the first ten natural numbers is $1^2 + 2^2 + ... + 10^2 = 385$.

Hence the difference between the square of the sum of the first ten natural numbers and the sum of the squares of the first ten natural numbers is 3025 - 385 = 2640.

[3]:
```python
def square(x):
    return x ** 2
def dif_square(n):
    return sum(range(n+1))**2 - sum(map(lambda x:x**2,range(n+1)))

print(dif_square(10))
```

```
2640
```

1.3 Armstrong Numbers

An Armstrong number is a number that is the sum of its own digits each raised to the power of the number of digits.

For example:

- 9 is an Armstrong number, because `9 = 9^1 = 9`
- 10 is *not* an Armstrong number, because `10 != 1^2 + 0^2 = 1`
- 153 is an Armstrong number, because: `153 = 1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153`
- 154 is *not* an Armstrong number, because: `154 != 1^3 + 5^3 + 4^3 = 1 + 125 + 64 = 190`

Write some code to determine whether a number is an Armstrong number.

[4]:
```python
def armstrong(number:int):
    sum = 0
    temp = number
    order = len(str(number))
    while temp > 0:
       digit = temp % 10
       sum += digit ** order
       temp //= 10
    return True if sum == number else False

print(armstrong(9926314))
```

```
False
```

1.4 Linked List

Implement a doubly linked list.

Like an array, a linked list is a simple linear data structure. Several common data types can be implemented using linked lists, like queues, stacks, and associative arrays.

A linked list is a collection of data elements called *nodes*. In a *singly linked list* each node holds a value and a link to the next node. In a *doubly linked list* each node also holds a link to the previous node.

You will write an implementation of a doubly linked list. Implement a Node to hold a value and pointers to the next and previous nodes. Then implement a List which holds references to the first and last node and offers an array-like interface for adding and removing items:

- `push` (*insert value at back*);
- `pop` (*remove value at back*);
- `shift` (*remove value at front*).
- `unshift` (*insert value at front*);

To keep your implementation simple, the tests will not cover error conditions. Specifically: `pop` or `shift` will never be called on an empty list.

If you want to know more about linked lists, check Wikipedia.

You also must implement functions to make it an iterator, and to make the function `len` work on your objects.

[5]:
```
class Node:
    def __init__(self, value, next=None, previous=None):
        self.value = value
        self.next = next
        self.previous = previous

class LinkedListIterator:
    def __init__(self, head):
        self.current = head

    def __iter__(self):
        return self

    def __next__(self):
        if self.current is not None:
            item = self.current.value
            self.current = self.current.next
            return item
        else:
            raise StopIteration

class LinkedList:
    def __init__(self):
        self.head = None
        self.tail = None

    def unshift(self, value):
        new_node = Node(value)
        if self.head is None:
            self.head = self.tail = new_node
        else:
            new_node.next = self.head
            self.head.previous = new_node
            self.head = new_node

    def pop(self):
        if self.tail is not None:
            if self.head == self.tail:
                self.head = self.tail = None
            else:
                last_node = self.tail
                self.tail = last_node.previous
                self.tail.next = None
    def shift(self):
        if self.head is not None:
            if self.head == self.tail:
                self.head = self.tail = None
            else:
                second_node = self.head.next
                second_node.previous = None
                self.head = second_node

    def push(self, value):
        new_node = Node(value)
        if self.head is None:
            self.head = self.tail = new_node
        else:
            new_node.previous = self.tail
            self.tail.next = new_node
            self.tail = new_node

    def __iter__(self):
        return LinkedListIterator(self.head)

    def __len__(self):
        current = self.head
        count = 0
        while current:
            count += 1
            current = current.next
        return count
```

[7]:
```
# Create a new linked list
my_list = LinkedList()

# Add elements to the list using unshift (insert at front)
my_list.unshift(3)
my_list.unshift(2)
my_list.unshift(1)

# Add elements to the list using push (insert at back)
my_list.push(4)
my_list.push(5)

# Display the length of the list
print("Length:", len(my_list))  # Output: Length: 5

# Display the elements in the list using iteration
print("Elements:", end=" ")
for idx, item in enumerate(my_list):
    print(item, end=" ")
    if idx < len(my_list) - 1:
        print("->", end=" ")
print()

# Remove elements from the list using shift (remove from front)
my_list.shift()
my_list.shift()

# Remove elements from the list using pop (remove from back)
my_list.pop()

# Display the length of the modified list
print("Length after modifications:", len(my_list))  # Output: Length after␣
↪modifications: 2

# Display the elements in the modified list using iteration
print("Modified elements:", end=" ")
for idx, item in enumerate(my_list):
    print(item, end=" ")
    if idx < len(my_list) - 1:
        print("->", end=" ")
print()
```

```
Length: 5
Elements: 1 -> 2 -> 3 -> 4 -> 5
Length after modifications: 2
Modified elements: 3 -> 4
```

1.5 List Ops

Implement basic list operations.

In functional languages list operations like `length`, `map`, and `reduce` are very common. Implement a series of basic list operations, without using existing functions.

Read tests to understand what is expected for each function.

[8]:
```
# Append elements from ys to xs
def append(xs, ys):
    if not xs:
        xs = []
        for num in ys:
            xs += [num]
    else:
        for num in ys:
            xs += [num]
    return xs
```

```python
# Concatenate multiple lists into a single list
def concat(lists):
    sum_list = []
    for li in lists:
        for item in li:
            sum_list += [item]
    return sum_list

# Filter elements from xs using a filtering function
def filter_clone(function, xs):
    filtered = []
    for val in xs:
        if function(val):
            filtered += [val]
    return filtered

# Calculate the length of the list xs
def length(xs):
    count = 0
    for _ in xs:
        count += 1
    return count

# Apply a mapping function to each element of xs
def map_clone(function, xs):
    mapped = []
    for val in xs:
        mapped += [function(val)]
    return mapped

# Perform a left fold operation on xs with an accumulator
def foldl(function, xs, acc):
    result = acc
    for val in xs:
        result = function(result, val)
    return result

# Perform a right fold operation on xs with an accumulator
def foldr(function, xs, acc):
    result = acc
    for idx in range(len(xs) - 1, -1, -1):
        result = function(xs[idx], result)
    return result

# Reverse the elements of xs
def reverse(xs):
    reversed_list = []
    for idx in range(len(xs) - 1, -1, -1):
        reversed_list += [xs[idx]]
    return reversed_list
```

[10]:
```
xs = [1, 2, 3, 4, 5]
ys = [6, 7, 8]
lists = [[9, 10], [11, 12, 13], [14]]

# Append
appended = append(xs.copy(), ys)
print("Appended:", appended)

# Concat
concatenated = concat(lists)
print("Concatenated:", concatenated)

# Filter
def is_even(num):
    return num % 2 == 0

filtered = filter_clone(is_even, xs)
print("Filtered:", filtered)

# Length
length_xs = length(xs)
print("Length of xs:", length_xs)

# Map
def square(num):
    return num ** 2

mapped = map_clone(square, xs)
print("Mapped:", mapped)

# Fold Left
def add(a, b):
    return a + b

folded_left = foldl(add, xs, 0)
print("Folded left:", folded_left)

# Fold Right
folded_right = foldr(add, xs, 0)
print("Folded right:", folded_right)

# Reverse
reversed_list = reverse(xs)
```

```python
print("Reversed:", reversed_list)
```

```
Appended: [1, 2, 3, 4, 5, 6, 7, 8]
Concatenated: [9, 10, 11, 12, 13, 14]
Filtered: [2, 4]
Length of xs: 5
Mapped: [1, 4, 9, 16, 25]
Folded left: 15
Folded right: 15
Reversed: [5, 4, 3, 2, 1]
```

1.6 Rotational Cipher

Create an implementation of the rotational cipher, also sometimes called the Caesar cipher.

The Caesar cipher is a simple shift cipher that relies on transposing all the letters in the alphabet using an integer key between `0` and `26`. Using a key of `0` or `26` will always yield the same output due to modular arithmetic. The letter is shifted for as many values as the value of the key.

The general notation for rotational ciphers is `ROT + <key>`. The most commonly used rotational cipher is `ROT13`.

A `ROT13` on the Latin alphabet would be as follows:

```
Plain:  abcdefghijklmnopqrstuvwxyz
Cipher: nopqrstuvwxyzabcdefghijklm
```

It is stronger than the Atbash cipher because it has 27 possible keys, and 25 usable keys.

Ciphertext is written out in the same formatting as the input including spaces and punctuation.

1.6.1 Examples

- ROT5 `omg` gives `trl`
- ROT0 `c` gives `c`
- ROT26 `Cool` gives `Cool`
- ROT13 `The quick brown fox jumps over the lazy dog.` gives `Gur dhvpx oebja sbk whzcf bire gur ynml qbt.`
- ROT13 `Gur dhvpx oebja sbk whzcf bire gur ynml qbt.` gives `The quick brown fox jumps over the lazy dog.`

[13]:
```python
def rotate(text, key):
    def shift_letter(letter, base, shift):
        return chr((ord(letter) - base + shift) % 26 + base)

    rotated_text = []
    for letter in text:
        if letter.isalpha():
            if letter.isupper():
                rotated_text.append(shift_letter(letter, ord('A'), key))
            else:
                rotated_text.append(shift_letter(letter, ord('a'), key))
        else:
            rotated_text.append(letter)

    return ''.join(rotated_text)


print(rotate("omg", 5))
print(rotate("c", 0))
print(rotate("Cool", 26))
print(rotate("The quick brown fox jumps over the lazy dog.", 13))
print(rotate("Gur dhvpx oebja sbk whzcf bire gur ynml qbt.", 13))
```

```
trl
c
Cool
Gur dhvpx oebja sbk whzcf bire gur ynml qbt.
The quick brown fox jumps over the lazy dog.
```

1.7 ETL

We are going to do the `Transform` step of an Extract-Transform-Load.

1.7.1 ETL

Extract-Transform-Load (ETL) is a fancy way of saying, "We have some crufty, legacy data over in this system, and now we need it in this shiny new system over here, so we're going to migrate this."

(Typically, this is followed by, "We're only going to need to run this once." That's then typically followed by much forehead slapping and moaning about how stupid we could possibly be.)

1.7.2 The goal

We're going to extract some scrabble scores from a legacy system.

The old system stored a list of letters per score:

- 1 point: "A", "E", "I", "O", "U", "L", "N", "R", "S", "T",
- 2 points: "D", "G",
- 3 points: "B", "C", "M", "P",
- 4 points: "F", "H", "V", "W", "Y",
- 5 points: "K",
- 8 points: "J", "X",
- 10 points: "Q", "Z",

The shiny new scrabble system instead stores the score per letter, which makes it much faster and easier to calculate the score for a word. It also stores the letters in lower-case regardless of the case of the input letters:

- "a" is worth 1 point.

- “b” is worth 3 points.
- “c” is worth 3 points.
- “d” is worth 2 points.
- Etc.

Your mission, should you choose to accept it, is to transform the legacy data format to the shiny new format.

[16]:
```python
def transform(legacy_data: dict) -> str:
    new_data = {}
    for score, letters in legacy_data.items():
        for letter in letters:
            new_data[letter.lower()] = score
    return new_data

# Legacy data in the format of points per letter
legacy_data = {
    1: ["A", "E", "I", "O", "U", "L", "N", "R", "S", "T"],
    2: ["D", "G"],
    3: ["B", "C", "M", "P"],
    4: ["F", "H", "V", "W", "Y"],
    5: ["K"],
    8: ["J", "X"],
    10: ["Q", "Z"]
}

# Transform the legacy data into the new format
new_data = transform(legacy_data)

# Print the new format data
for letter, score in new_data.items():
    print(f"'{letter}' is worth {score} point{'s' if score > 1 else ''}.")
```

```
'a' is worth 1 point.
'e' is worth 1 point.
'i' is worth 1 point.
'o' is worth 1 point.
'u' is worth 1 point.
'l' is worth 1 point.
'n' is worth 1 point.
'r' is worth 1 point.
's' is worth 1 point.
't' is worth 1 point.
'd' is worth 2 points.
'g' is worth 2 points.
'b' is worth 3 points.
'c' is worth 3 points.
'm' is worth 3 points.
'p' is worth 3 points.
```

```
'f' is worth 4 points.
'h' is worth 4 points.
'v' is worth 4 points.
'w' is worth 4 points.
'y' is worth 4 points.
'k' is worth 5 points.
'j' is worth 8 points.
'x' is worth 8 points.
'q' is worth 10 points.
'z' is worth 10 points.
```

1.8 Say

Given a number from 0 to 999,999,999,999, spell out that number in English.

1.8.1 Step 1

Handle the basic case of 0 through 99.

If the input to the program is `22`, then the output should be `'twenty-two'`.

Your program should complain loudly if given a number outside the blessed range.

Some good test cases for this program are:

- 0
- 14
- 50
- 98
- -1
- 100

Extension If you're on a Mac, shell out to Mac OS X's `say` program to talk out loud. If you're on Linux or Windows, eSpeakNG may be available with the command `espeak`.

1.8.2 Step 2

Implement breaking a number up into chunks of thousands.

So `1234567890` should yield a list like 1, 234, 567, and 890, while the far simpler `1000` should yield just 1 and 0.

The program must also report any values that are out of range.

1.8.3 Step 3

Now handle inserting the appropriate scale word between those chunks.

So `1234567890` should yield `'1 billion 234 million 567 thousand 890'`

The program must also report any values that are out of range. It's fine to stop at "trillion".

1.8.4 Step 4

Put it all together to get nothing but plain English.

`12345` should give `twelve thousand three hundred forty-five`.

The program must also report any values that are out of range.

Extensions Use *and* (correctly) when spelling out the number in English:

- 14 becomes "fourteen".
- 100 becomes "one hundred".
- 120 becomes "one hundred and twenty".
- 1002 becomes "one thousand and two".
- 1323 becomes "one thousand three hundred and twenty-three".

[17]:
```
def say(number):
    if number == 0:
        return "zero"

    ones = ["", "one", "two", "three", "four", "five", "six", "seven", "eight",␣
↪"nine", "ten",
            "eleven", "twelve", "thirteen", "fourteen", "fifteen", "sixteen",␣
↪"seventeen", "eighteen", "nineteen"]

    tens = ["", "", "twenty", "thirty", "forty", "fifty", "sixty", "seventy",␣
↪"eighty", "ninety"]

    if number < 0 or number >= 1e12:
        raise ValueError("Number out of range")

    if number < 20:
        return ones[number]
    elif number < 100:
        return tens[number // 10] + ("-" + ones[number % 10] if number % 10 !=␣
↪0 else "")
    elif number < 1000:
        return ones[number // 100] + " hundred" + ((" and " + say(number %␣
↪100)) if number % 100 != 0 else "")
    else:
        for i, scale in enumerate(["thousand", "million", "billion"]):
            if number < 1000 ** (i + 2):
                return say(number // 1000 ** (i + 1)) + " " + scale + ((" " +␣
↪say(number % 1000 ** (i + 1))) if number % 1000 ** (i + 1) != 0 else "")

print(say(1185408))
```

```
one million one hundred and eighty-five thousand four hundred and eight
```

1.9 Run Length Encoding

Implement run-length encoding and decoding.

Run-length encoding (RLE) is a simple form of data compression, where runs (consecutive data elements) are replaced by just one data value and count.

For example we can represent the original 53 characters with only 13.

`"WWWWWWWWWWWWBWWWWWWWWWWWWBBBWWWWWWWWWWWWWWWWWWWWWWWWB"  ->  "12WB12W3B24WB"`

RLE allows the original data to be perfectly reconstructed from the compressed data, which makes it a lossless data compression.

`"AABCCCDEEEE"  ->  "2AB3CD4E"  ->  "AABCCCDEEEE"`

For simplicity, you can assume that the unencoded string will only contain the letters A through Z (either lower or upper case) and whitespace. This way data to be encoded will never contain any numbers and numbers inside data to be decoded always represent the count for the following character.

[20]:
```python
def encode(string: str) -> str:
    count=1
    code=''
    char = string[0]
    for idx in range(1, len(string)):
        if string[idx] == char:
            count +=1
        else:
            code += f'{count if count > 1 else ""}{char}'
            count = 1
            char = string[idx]
    code += f'{count if count > 1 else ""}{char}'
    return code

def decode(string: str) -> str:
    if not string:
        return string
    else:
        num = string[0]
        code = ""
        for idx in range(1, len(string)):
            if string[idx].isdigit():
                num += string[idx]
            else:
                if num:
                    code += f"{string[idx]}"* int(num)
                else:
                    code += f"{string[idx]}"
                num = ''
        return code
```

```
print(encode('zzz ZZ  zZ'))
print(decode(encode('zzz ZZ  zZ')))
```

```
3z 2Z2 zZ
zzz ZZ  zZ
```

1.10 json.dumps clone

[ ]:
```
def custom_dumps(data: dict | list) ->str:
    if isinstance(data, (dict, list)):
        json_string = json_encode(data)
    else:
        raise TypeError("data must be a dictionary or a list.")
    return json_string

def json_encode(data):
    if isinstance(data, dict):
        json_string = "{"
        for key, value in data.items():
            json_string += f'"{key}":{json_encode(value)},'
        json_string = json_string[:-1] + "}"
    elif isinstance(data, list):
        json_string = "["
        for item in data:
            json_string += f"{json_encode(item)},"
        json_string = json_string[:-1] + "]"
    elif isinstance(data, str):
        json_string = f'"{data}"'
    elif isinstance(data, (int, float)):
        json_string = str(data)
    else:
        json_string = "null"
    return json_string

data = {'name': 'John', 'age': 30, 'city': 'New York'}
json_string = custom_dumps(data)
print(json_string)
```