1. Create a class Sudoku that takes a string as an argument. The string will contain the numbers of a regular 9x9 sudoku board left to right and top to bottom, with zeros filling up the empty cells.

Attributes

An instance of the class Sudoku will have one attribute:

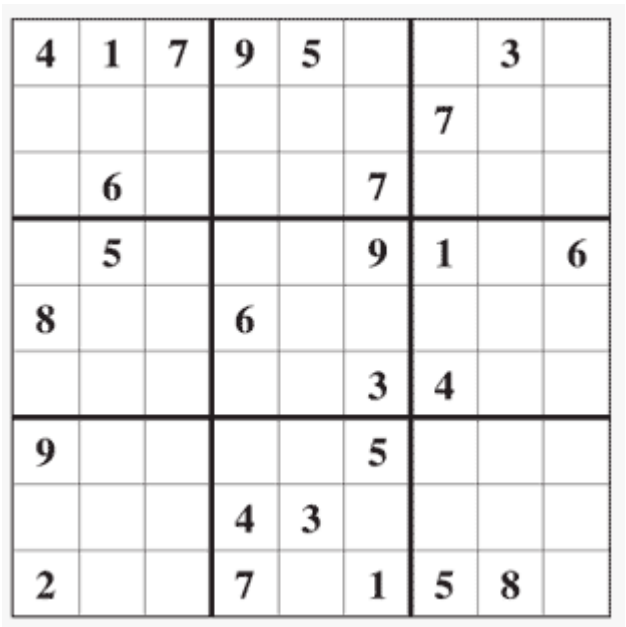
- board: a list representing the board, with sublits for each row, with the numbers as integers. Empty cell represented with 0.

Methods

An instance of the class Sudoku wil have three methods:

- get_row(n): will return the row in position n.
- get_col(n): will return the column in position n.
- get_sqr([n, m]): will return the square in position n if only one argument is given, and the square to which the cell in position (n, m) belongs to if two arguments are given.

Example



4	1	7	9	5			3	
						7		
	6				7			
	5				9	1		6
8			6					
					3	4		
9					5			
			4	3				
2			7		1	5	8	

game = Sudoku("417950030000000700060007000050009106800600000000003400900005000000430000200701580")

```
game.board    [
  [4, 1, 7, 9, 5, 0, 0, 3, 0],
  [0, 0, 0, 0, 0, 0, 7, 0, 0],
  [0, 6, 0, 0, 0, 7, 0, 0, 0],
  [0, 5, 0, 0, 0, 9, 1, 0, 6],
  [8, 0, 0, 6, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 3, 4, 0, 0],
  [9, 0, 0, 0, 0, 5, 0, 0, 0],
  [0, 0, 0, 4, 3, 0, 0, 0, 0],
  [2, 0, 0, 7, 0, 1, 5, 8, 0]
]

game.get_row(0)    [4, 1, 7, 9, 5, 0, 0, 3, 0]
game.get_col(8)    [0, 0, 0, 6, 0, 0, 0, 0, 0]
game.get_sqr(1)    [9, 5, 0, 0, 0, 0, 0, 0, 7]
game.get_sqr(1, 8)    [0, 3, 0, 7, 0, 0, 0, 0, 0]
game.get_sqr(8, 3)    [0, 0, 5, 4, 3, 0, 7, 0, 1]
```

2. The function input is two non-empty linked lists representing two non-negative integers. The digits are stored in reverse order, and each of their nodes contains a single digit. Add the two numbers and return the sum as a linked list, in which the digits are also stored in reversed order. The class ListNode, building block of the linked list, is defined in the Tests tab.

Class definition

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next
```

Examples

```
lt1 = ListNode(2)
lt1.add_data([4, 3])
lt2 = ListNode(5)
lt2.add_data([6, 4])
# print(lt1.get_data())    # [2, 4, 3]
# print(lt2.get_data())    # [5, 6, 4]
# print(342 + 465)         # 807
add_two_numbers(lt1, lt2).get_data()    [7, 0, 8]

lt1 = ListNode(0)
lt2 = ListNode(0)
# print(lt1.get_data())    # [0]
```

```
# print(lt2.get_data())    # [0]
# print(0 + 0)            # 0
add_two_numbers(lt1, lt2).get_data()     [0]

lt1 = ListNode(9)
lt1.add_data([9,9,9,9,9,9])
lt2 = ListNode(9)
lt2.add_data([9,9,9])
# print(lt1.get_data())    # [9, 9, 9, 9, 9, 9, 9]
# print(lt2.get_data())    # [9, 9, 9, 9]
# print(9999999 + 9999)    # 10009998
add_two_numbers(lt1, lt2).get_data()     [8, 9, 9, 9, 0, 0, 0, 1]
```

3. Write a class called CoffeeShop, which has three instance variables:

1. name : a string (basically, of the shop)
2. menu : a list of items (of dict type), with each item containing the item (name of the item), type (whether a food or a drink) and price.
3. orders : an empty list

and seven methods:

1. add_order: adds the name of the item to the end of the orders list if it exists on the menu, otherwise, return "This item is currently unavailable!"
2. fulfill_order: if the orders list is not empty, return "The {item} is ready!". If the orders list is empty, return "All orders have been fulfilled!"
3. list_orders: returns the item names of the orders taken, otherwise, an empty list.
4. due_amount: returns the total amount due for the orders taken.
5. cheapest_item: returns the name of the cheapest item on the menu.
6. drinks_only: returns only the item names of type drink from the menu.
7. food_only: returns only the item names of type food from the menu.

IMPORTANT: Orders are fulfilled in a FIFO (first-in, first-out) order.

Examples

```
tcs.add_order("hot cocoa")     "This item is currently unavailable!"
# Tesha's coffee shop does not sell hot cocoa
tcs.add_order("iced tea")     "This item is currently unavailable!"
# specifying the variant of "iced tea" will help the process

tcs.add_order("cinnamon roll")      "Order added!"
tcs.add_order("iced coffee")     "Order added!"
tcs.list_orders     ["cinnamon roll", "iced coffee"]
```

all items of the current order

tcs.due_amount() 2.17

tcs.fulfill_order() "The cinnamon roll is ready!"
tcs.fulfill_order() "The iced coffee is ready!"
tcs.fulfill_order() "All orders have been fulfilled!"
all orders have been presumably served

tcs.list_orders() []
an empty list is returned if all orders have been exhausted

tcs.due_amount() 0.0
no new orders taken, expect a zero payable

tcs.cheapest_item() "lemonade"
tcs.drinks_only() ["orange juice", "lemonade", "cranberry juice", "pineapple juice", "lemon iced tea", "vanilla chai latte", "hot chocolate", "iced coffee"]
tcs.food_only() ["tuna sandwich", "ham and cheese sandwich", "bacon and egg", "steak", "hamburger", "cinnamon roll"]

4. In this challenge, write a funcion loneliest_number to find the last Lonely number inside a sequence. A number is Lonely if the distance from its closest Prime sets a new record of the sequence.

Sequence = from 0 to 3

Any number lower than 3 doesn't have a Prime preceeding it...
...so that you'll consider only its next closest Prime.

0 has distance 2 from its closest Prime (2)
It's a new record! 0 It's the first lonely number of the sequence
1 has distance 1 from its closest Prime (2)
2 has distance 1 from 3
3 has distance 1 from 2

The sequence 0 to 3 has only one Lonely number: 0

Sequence = Numbers from 5 to 10

5 has distance 2 from its closest Prime (3 or 7)
It's a new record! 5 It's the first lonely number of the sequence
6 has distance 1 from 5 or 7
7 has distance 2 from 5
8 has distance 1 from 7

9 has distance 2 from 7 or 11
10 has distance 1 from 11

The sequence 5 to 10 has only one Lonely number: 5

Sequence = Numbers from 19 to 24

19 has distance 2 from its closest Prime (17)
It's a new record! 19 It's the first lonely number of the sequence
20 has distance 1 from 19
21 has distance 2 from 5
22 has distance 1 from 23
23 has distance 4 from 17 or 29
It's a new record! 23 is the second lonely number of the sequence
24 has distance 1 from 23

The sequence 19 to 24 has two Lonely numbers: 19 and 23

The function loneliest_number must accept two integers lo and hi being the inclusive bounds of the sequence to analyze, and returns a dictionay (dict) object with the following keys and values:

- number: is the last Lonely number found in the given sequence;
- distance: is the distance of the number from its closest Prime;
- closest: is the Prime closest to number (if two Primes are equally distant from number, return the higher Prime).

Examples

```
loneliest_number(0, 22)    {
  number: 0, distance: 2, closest: 2
}

loneliest_number(8, 123)    {
  number: 53, distance: 6, closest: 59
}

loneliest_number(938, 1190)    {
  number: 1140, distance: 11, closest: 1151
}

loneliest_number(120, 1190)    {
  number: 211, distance: 12, closest: 223
}
```

5. Implement a class Selfie that can store the current state of the object in the form of binary string. It can take multiple pictures and then recover to a state it was before. During testing an object will be provided with new attributes and their values. It will store its state. Then the values will be changed. Then it will be given new attributes. It will store its state again. It will be repeated few times.

Later the states of the object will be recovered given an index. The return value should be a new Selfie with the requested historic state and the state history of the new object should be updated with a copy of current object's state history.

The object also knows how many states it has stored. If the index is not within the range of stored states, the object stays as is. If the argument is invalid, n < 0 or n >= self.n_states(), the current object (or a copy thereof) should be returned.

Examples

```
p = Selfie()
p.x = 2
p.save_state()
p.x = 5
p = p.recover_state(0)
p.x     2
```