1. Two Sum

Given an array of integers nums and an integer target, return indices of the two numbers

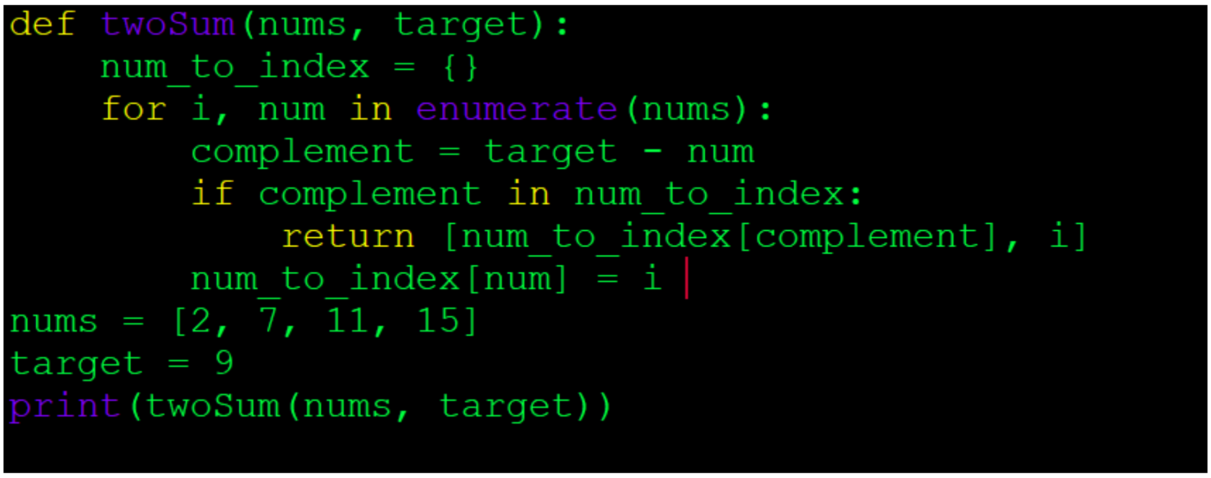
such that they add up to target.

You may assume that each input would have exactly one solution, and you may not use

the same element twice.

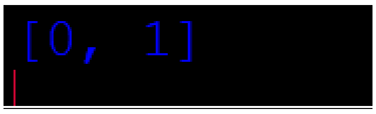
You can return the answer in any order.

Code:



```
def twoSum(nums, target):
    num_to_index = {}
    for i, num in enumerate(nums):
        complement = target - num
        if complement in num_to_index:
            return [num_to_index[complement], i]
        num_to_index[num] = i
nums = [2, 7, 11, 15]
target = 9
print(twoSum(nums, target))
```

Output:



```
[0, 1]
```

2. Add Two Numbers

You are given two non-empty linked lists representing two non-negative integers. The

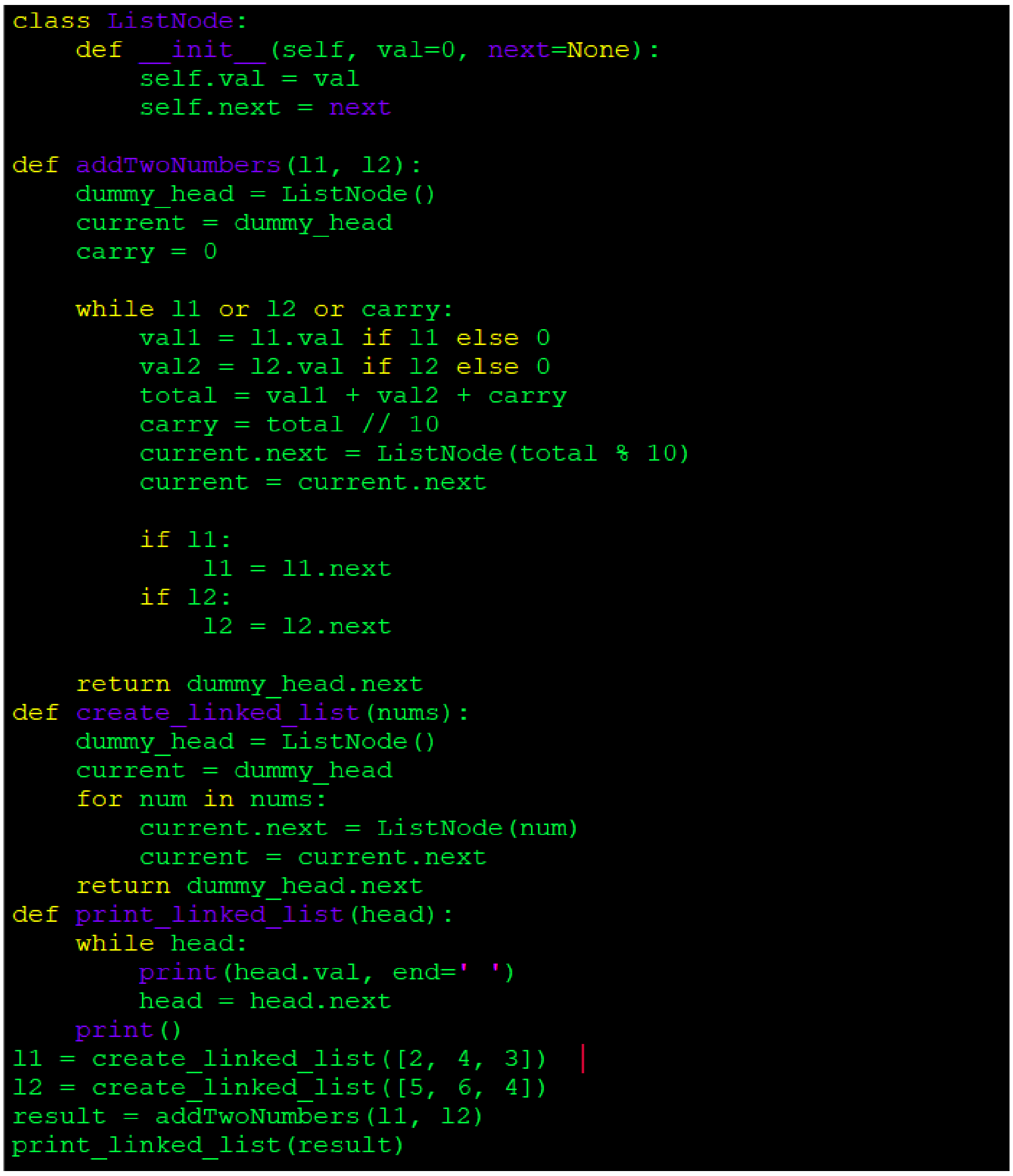
digits are stored in reverse order, and each of their nodes contains a single digit. Add the

two numbers and return the sum as a linked list.

You may assume the two numbers do not contain any leading zero, except the number 0

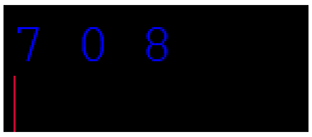
itself.

Code:



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

def addTwoNumbers(l1, l2):
    dummy_head = ListNode()
    current = dummy_head
    carry = 0

    while l1 or l2 or carry:
        val1 = l1.val if l1 else 0
        val2 = l2.val if l2 else 0
        total = val1 + val2 + carry
        carry = total // 10
        current.next = ListNode(total % 10)
        current = current.next

        if l1:
            l1 = l1.next
        if l2:
            l2 = l2.next

    return dummy_head.next
def create_linked_list(nums):
    dummy_head = ListNode()
    current = dummy_head
    for num in nums:
        current.next = ListNode(num)
        current = current.next
    return dummy_head.next
def print_linked_list(head):
    while head:
        print(head.val, end=' ')
        head = head.next
    print()
l1 = create_linked_list([2, 4, 3])
l2 = create_linked_list([5, 6, 4])
result = addTwoNumbers(l1, l2)
print_linked_list(result)
```

Output:



```
7 0 8
```

3. Longest Substring without Repeating Characters

Given a string s, find the length of the longest substring without repeating characters.

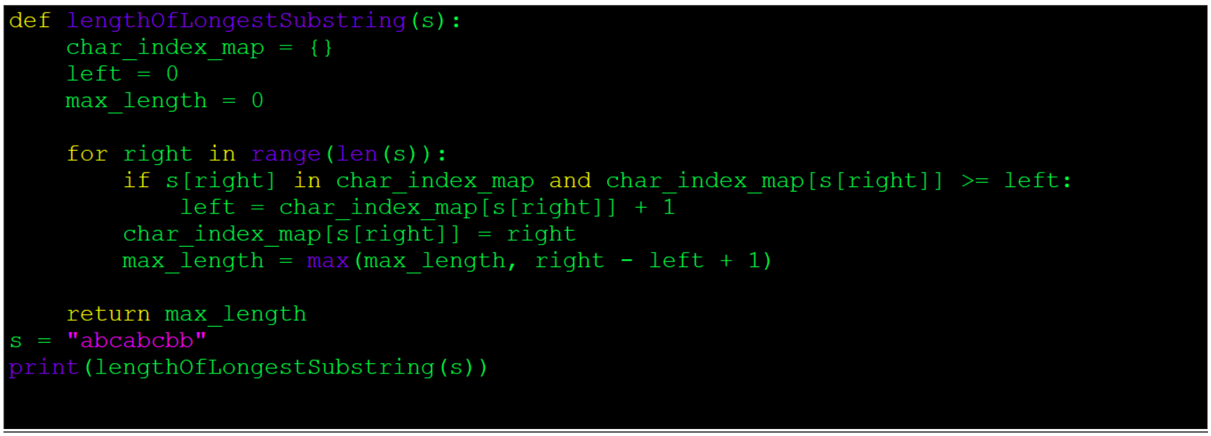
Example 1:

Input: s = "abcabcbb"

Output: 3

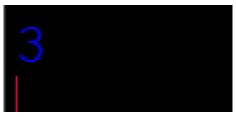
Explanation: The answer is "abc", with the length of 3.

Code:



```python
def lengthOfLongestSubstring(s):
    char_index_map = {}
    left = 0
    max_length = 0

    for right in range(len(s)):
        if s[right] in char_index_map and char_index_map[s[right]] >= left:
            left = char_index_map[s[right]] + 1
        char_index_map[s[right]] = right
        max_length = max(max_length, right - left + 1)

    return max_length
s = "abcabcbb"
print(lengthOfLongestSubstring(s))
```

Output:



```
3
```

4. Median of Two Sorted Arrays

Given two sorted arrays nums1 and nums2 of size m and n respectively, return the

median of the two sorted arrays.

The overall run time complexity should be O(log (m+n)).

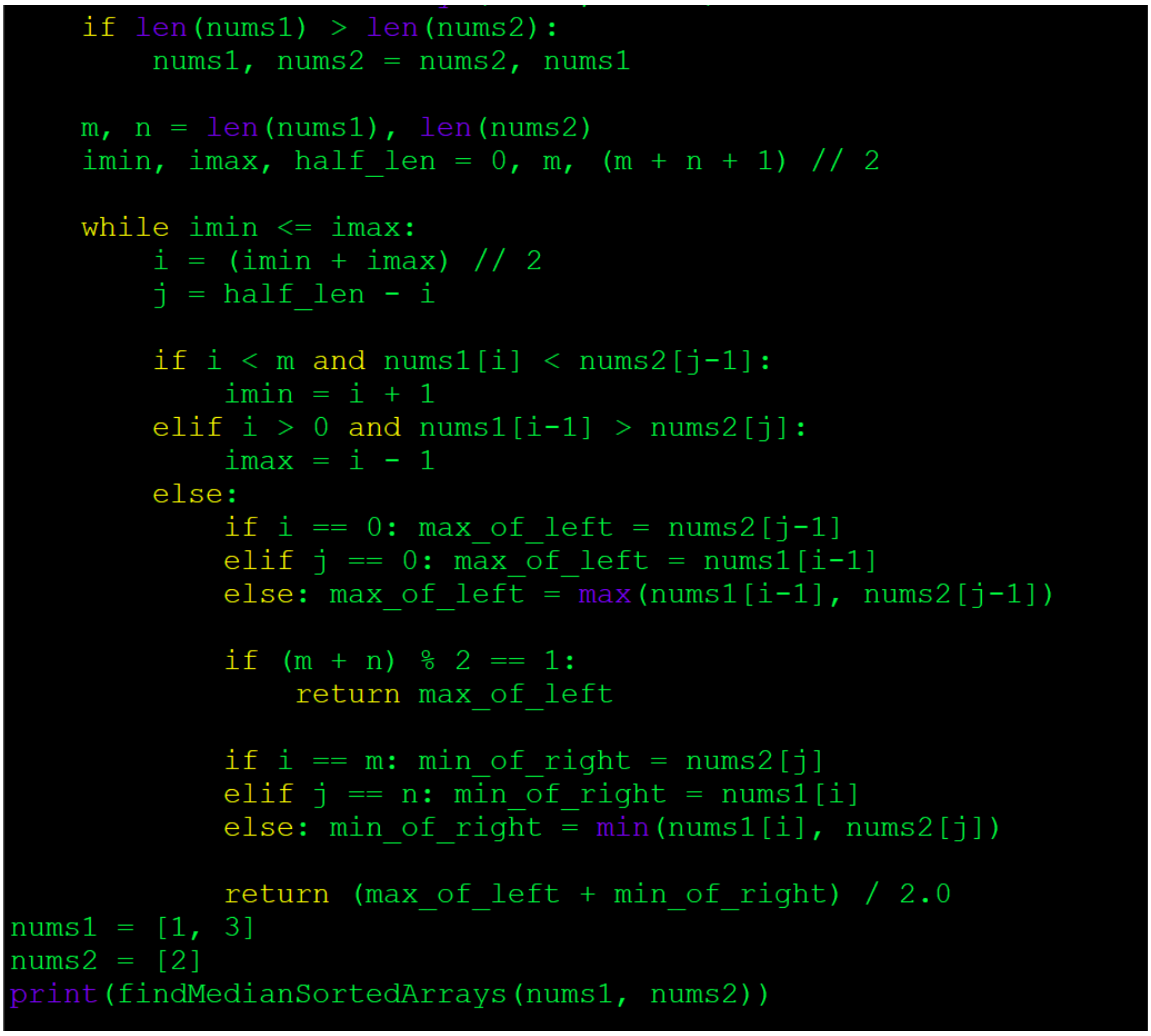
Example 1:

Input: nums1 = [1,3], nums2 = [2]

Output: 2.00000

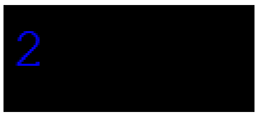
Explanation: merged array = [1,2,3] and median is 2.

<u>Code:</u>



```
    if len(nums1) > len(nums2):
        nums1, nums2 = nums2, nums1

    m, n = len(nums1), len(nums2)
    imin, imax, half_len = 0, m, (m + n + 1) // 2

    while imin <= imax:
        i = (imin + imax) // 2
        j = half_len - i

        if i < m and nums1[i] < nums2[j-1]:
            imin = i + 1
        elif i > 0 and nums1[i-1] > nums2[j]:
            imax = i - 1
        else:
            if i == 0: max_of_left = nums2[j-1]
            elif j == 0: max_of_left = nums1[i-1]
            else: max_of_left = max(nums1[i-1], nums2[j-1])

            if (m + n) % 2 == 1:
                return max_of_left

            if i == m: min_of_right = nums2[j]
            elif j == n: min_of_right = nums1[i]
            else: min_of_right = min(nums1[i], nums2[j])

            return (max_of_left + min_of_right) / 2.0
nums1 = [1, 3]
nums2 = [2]
print(findMedianSortedArrays(nums1, nums2))
```

<u>Output:</u>



```
2
```

5. Longest Palindromic Substring

Given a string s, return the longest palindromic substring in s.

Example 1:

Input: s = "babad"

Output: "bab"

Explanation: "aba" is also a valid answer.

Example 2:

Input: s = "cbbd"

Output: "bb"

Constraints:

- **1 <= s.length <= 1000**
- **s consist of only digits and English letters.**

Code:

```python
def longestPalindrome(s):
    if not s:
        return ""

    start, end = 0, 0

    for i in range(len(s)):
        len1 = expandAroundCenter(s, i, i)
        len2 = expandAroundCenter(s, i, i + 1)
        max_len = max(len1, len2)

        if max_len > end - start:
            start = i - (max_len - 1) // 2
            end = i + max_len // 2

    return s[start:end + 1]

def expandAroundCenter(s, left, right):
    while left >= 0 and right < len(s) and s[left] == s[right]:
        left -= 1
        right += 1
    return right - left - 1
s1 = "babad"
s2 = "cbbd"
print(longestPalindrome(s1))
print(longestPalindrome(s2))
```

Output:

```
aba
bb
```

6. Zigzag Conversion

The string "PAYPALISHIRING" is written in a zigzag pattern on a given number of

rows like this: (you may want to display this pattern in a fixed font for better legibility)

P A H N

A P L S I I G

Y I R

And then read line by line: "PAHNAPLSIIGYIR"

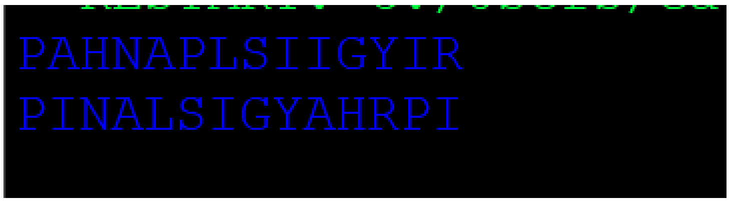
Write the code that will take a string and make this conversion given a number of rows:

string convert(string s, int numRows);

Code:

```python
def convert(s, numRows):
    if numRows == 1 or numRows >= len(s):
        return s

    rows = [''] * numRows
    current_row = 0
    going_down = False

    for char in s:
        rows[current_row] += char
        if current_row == 0 or current_row == numRows - 1:
            going_down = not going_down
        current_row += 1 if going_down else -1

    return ''.join(rows)
s = "PAYPALISHIRING"
numRows = 3
print(convert(s, numRows))

numRows = 4
print(convert(s, numRows))
```

<u>Output:</u>



```
PAHNAPLSIIGYIR
PINALSIGYAHRPI
```

7. Reverse Integer

Given a signed 32-bit integer x, return x with its digits reversed. If reversing x causes the

value to go outside the signed 32-bit integer range [-231, 231 - 1], then return 0.

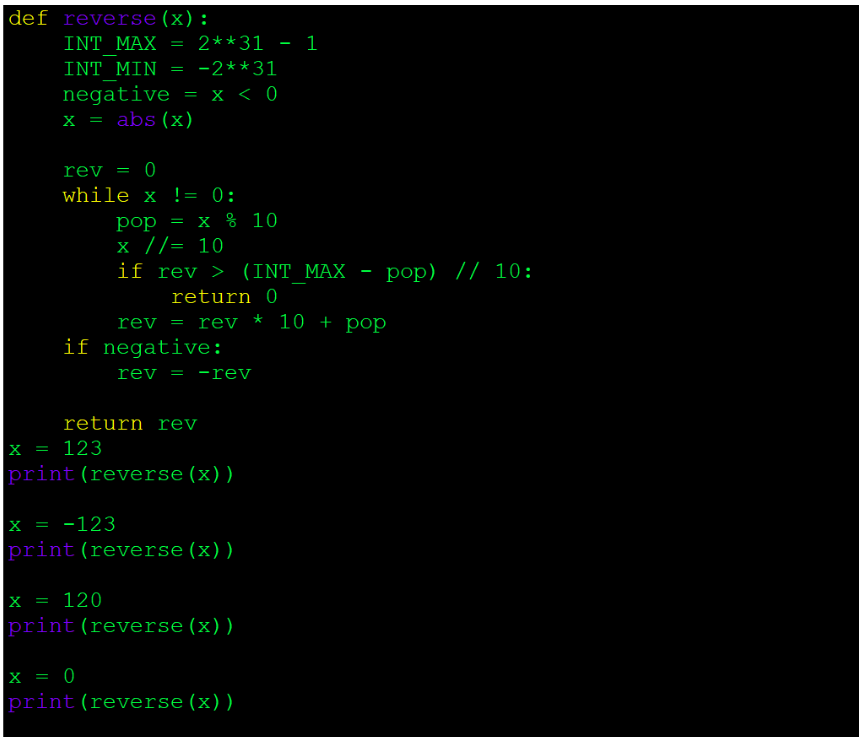
Assume the environment does not allow you to store 64-bit integers (signed or unsigned).

Example 1:

Input: x = 123

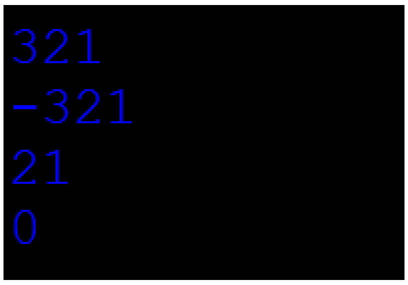
Output: 321

<u>Code:</u>



```python
def reverse(x):
    INT_MAX = 2**31 - 1
    INT_MIN = -2**31
    negative = x < 0
    x = abs(x)

    rev = 0
    while x != 0:
        pop = x % 10
        x //= 10
        if rev > (INT_MAX - pop) // 10:
            return 0
        rev = rev * 10 + pop
    if negative:
        rev = -rev

    return rev
x = 123
print(reverse(x))

x = -123
print(reverse(x))

x = 120
print(reverse(x))

x = 0
print(reverse(x))
```

<u>Output:</u>



```
321
-321
21
0
```

8. String to Integer (atoi)

Implement the myAtoi(string s) function, which converts a string to a 32-bit signed

integer (similar to C/C++'s atoi function).

The algorithm for myAtoi(string s) is as follows:

1. Read in and ignore any leading whitespace.

2. Check if the next character (if not already at the end of the string) is '-' or '+'. Read

this character in if it is either. This determines if the final result is negative or

positive respectively. Assume the result is positive if neither is present.

3. Read in next the characters until the next non-digit character or the end of the

input is reached. The rest of the string is ignored.

4. Convert these digits into an integer (i.e. "123" -> 123, "0032" -> 32). If no digits

were read, then the integer is 0. Change the sign as necessary (from step 2).

5. If the integer is out of the 32-bit signed integer range [-231, 231 - 1], then clamp

the integer so that it remains in the range. Specifically, integers less than -231

should be clamped to -231, and integers greater than 231 - 1 should be clamped to

231 - 1.

6. Return the integer as the final result.

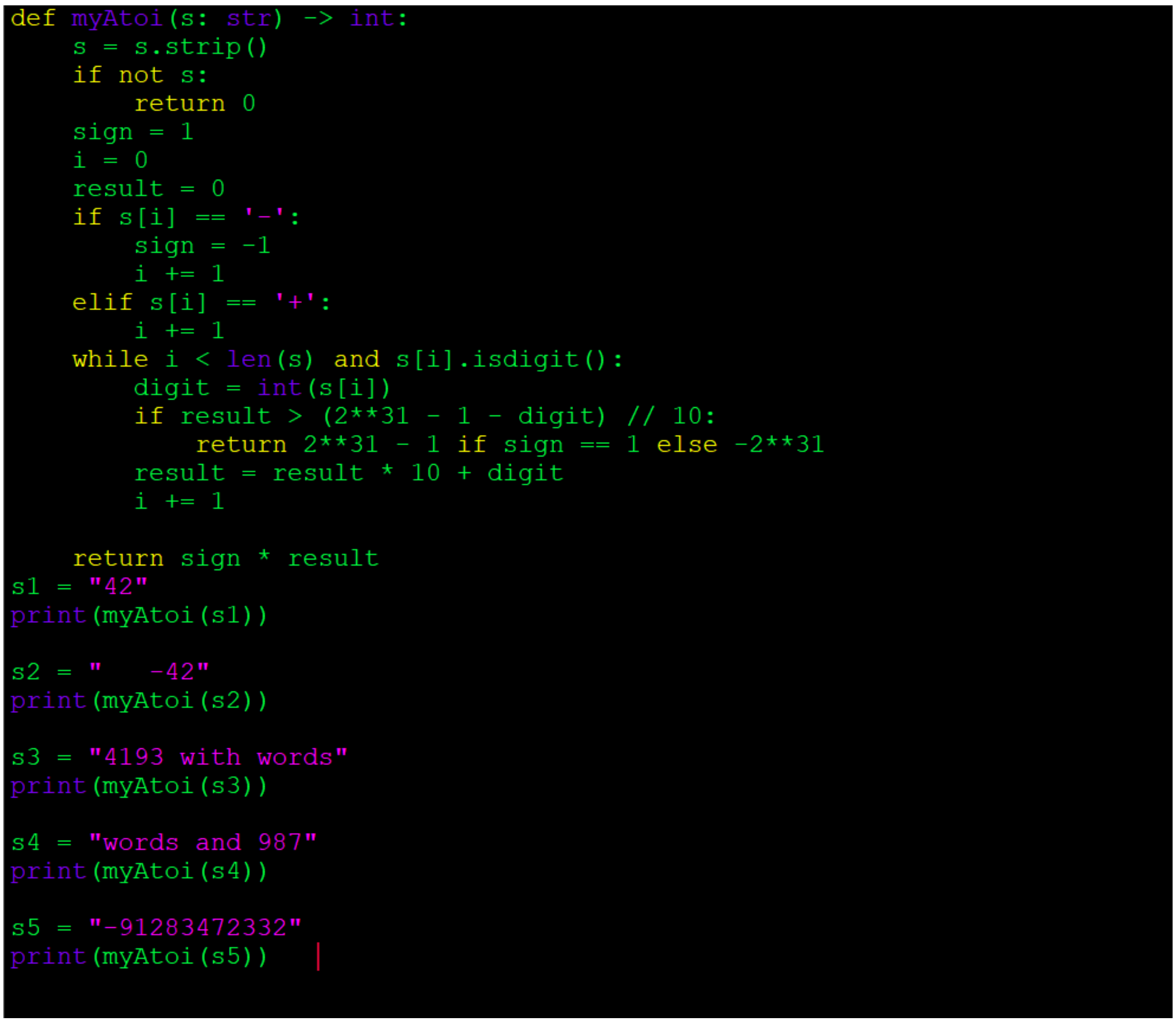
Note:

● Only the space character ' ' is considered a whitespace character.

● Do not ignore any characters other than the leading whitespace or the rest of the

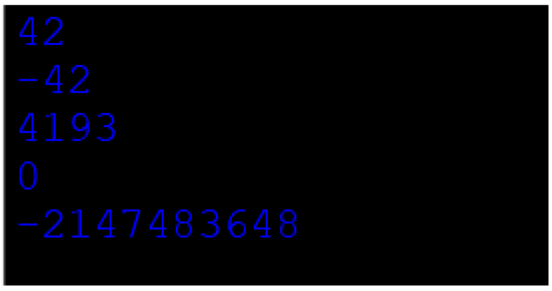
string after the digits.

<u>Code:</u>



```python
def myAtoi(s: str) -> int:
    s = s.strip()
    if not s:
        return 0
    sign = 1
    i = 0
    result = 0
    if s[i] == '-':
        sign = -1
        i += 1
    elif s[i] == '+':
        i += 1
    while i < len(s) and s[i].isdigit():
        digit = int(s[i])
        if result > (2**31 - 1 - digit) // 10:
            return 2**31 - 1 if sign == 1 else -2**31
        result = result * 10 + digit
        i += 1

    return sign * result
s1 = "42"
print(myAtoi(s1))

s2 = "   -42"
print(myAtoi(s2))

s3 = "4193 with words"
print(myAtoi(s3))

s4 = "words and 987"
print(myAtoi(s4))

s5 = "-91283472332"
print(myAtoi(s5))
```

Output:



```
42
-42
4193
0
-2147483648
```

9. Palindrome Number

Given an integer x, return true if x is a palindrome, and false otherwise.

Example 1:

Input: x = 121

Output: true

Explanation: 121 reads as 121 from left to right and from right to left.

Example 2:

Input: x = -121

Output: false

Explanation: From left to right, it reads -121. From right to left, it becomes 121-.

Therefore it is not a palindrome.

Example 3:

Input: x = 10

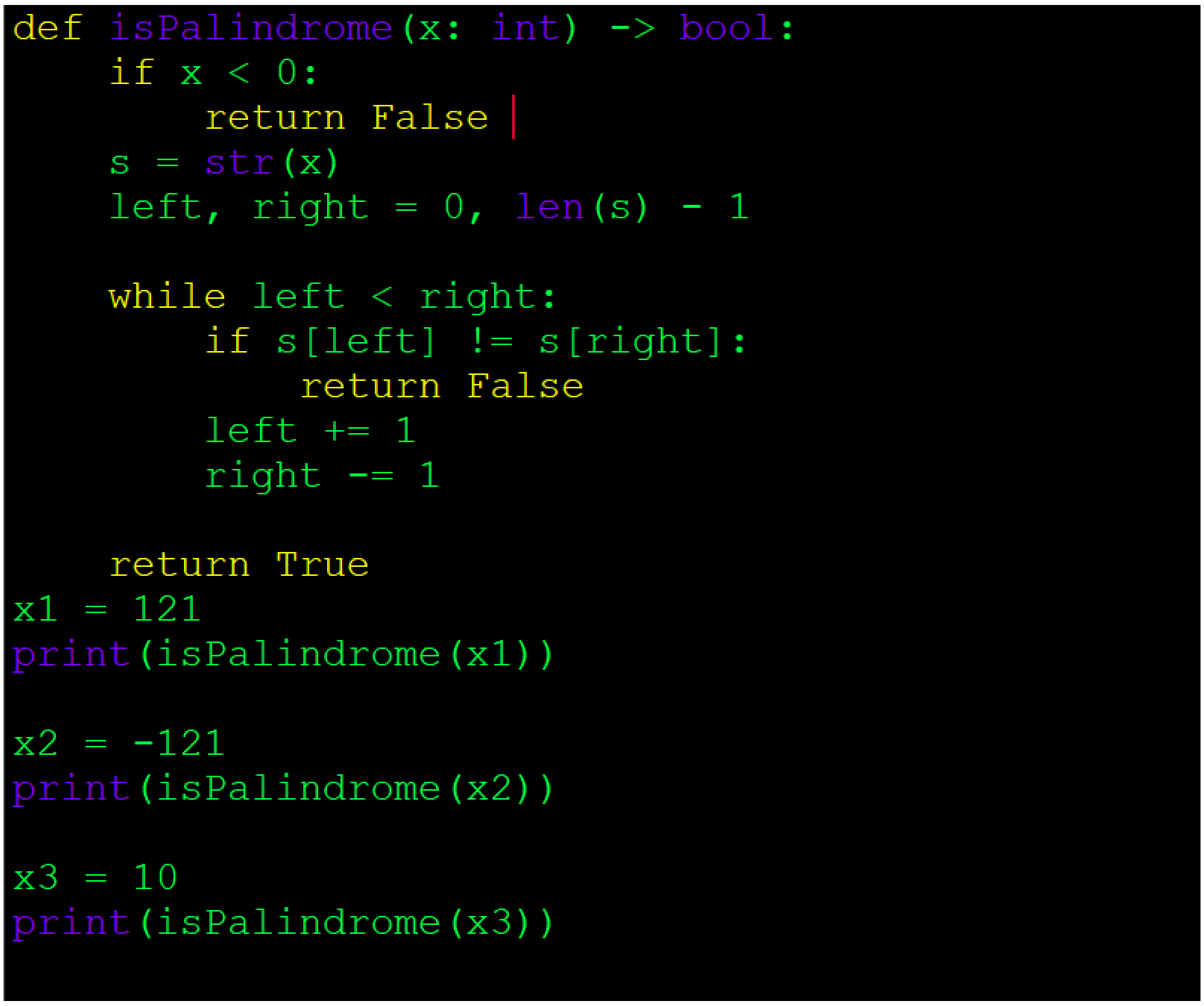
Output: false

Explanation: Reads 01 from right to left. Therefore it is not a palindrome.

Constraints:

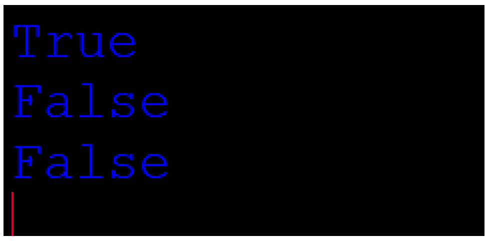
- **-231 <= x <= 231 – 1**

Code:



```python
def isPalindrome(x: int) -> bool:
    if x < 0:
        return False
    s = str(x)
    left, right = 0, len(s) - 1

    while left < right:
        if s[left] != s[right]:
            return False
        left += 1
        right -= 1

    return True
x1 = 121
print(isPalindrome(x1))

x2 = -121
print(isPalindrome(x2))

x3 = 10
print(isPalindrome(x3))
```

Output:



```
True
False
False
```

10. Regular Expression Matching

Given an input string s and a pattern p, implement regular expression matching with

support for '.' and '*' where:

● '.' Matches any single character.

- **'*' Matches zero or more of the preceding element.**

The matching should cover the entire input string (not partial).

Example 1:

Input: s = "aa", p = "a"

Output: false

Explanation: "a" does not match the entire string "aa".

Code:

```
def isMatch(s: str, p: str) -> bool:
    m, n = len(s), len(p)
    dp = [[False] * (n + 1) for _ in range(m + 1)]
    dp[0][0] = True
    for j in range(1, n + 1):
        if p[j - 1] == '*':
            dp[0][j] = dp[0][j - 2]
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if p[j - 1] == '.' or p[j - 1] == s[i - 1]:
                dp[i][j] = dp[i - 1][j - 1]
            elif p[j - 1] == '*':
                dp[i][j] = dp[i][j - 2]
                if p[j - 2] == '.' or p[j - 2] == s[i - 1]:
                    dp[i][j] = dp[i][j] or dp[i - 1][j]
            else:
                dp[i][j] = False

    return dp[m][n]
s = "aa"
p = "a"
print(isMatch(s, p))
```

Output:

```
False
```