



Look-and-Say Sequence

In this lesson, you will learn how to generate the next term of the Look-and-Say sequence in Python.

We'll cover the following ^

- Implementation
- Explanation

In this lesson, we will be considering the so-called “Look-and-Say” sequence. The first few terms of the sequence are:

1, 11, 21, 1211, 111221, 312211, 13112221, 1113213211, ...

To generate a member of the sequence from the previous member, read off the digits of the previous member and record the count of the number of digits in groups of the same digit.

For example, 1 is read off as one 1 which implies that the count of 1 is one. As 1 is read off as “one 1”, the next sequence will be written as 11 where 1 replaces one. Now 11 is read off as “two 1s” as the count of “1” is two in this term. Hence, the next term in the sequence becomes 21.

Have a look at the slides below where we generate this sequence up to the fifth term.



Look-and-Say Sequence : First Term

1
↓
one 1

1 of 5

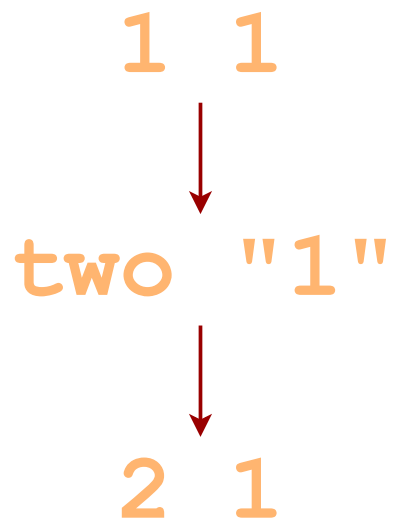
Look-and-Say Sequence : Second Term

1
↓
one "1"
↓
1 1

2 of 5

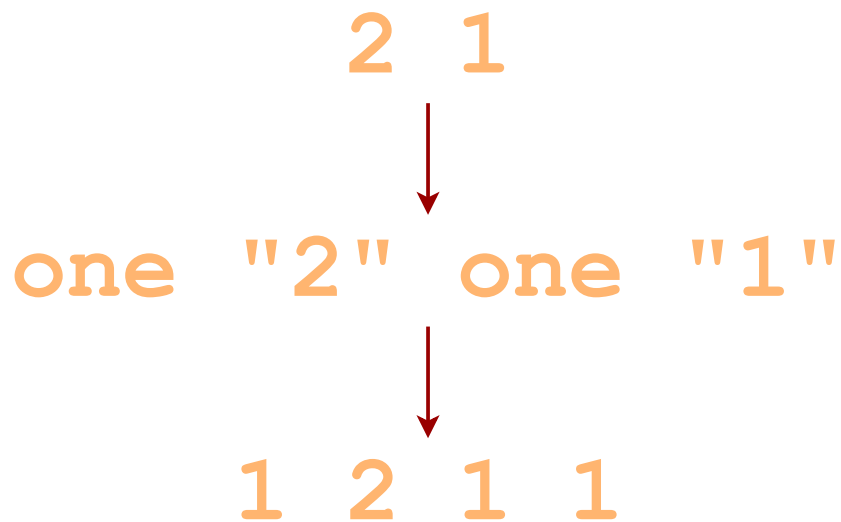


Look-and-Say Sequence : Third Term



3 of 5

Look-and-Say Sequence : Fourth Term



4 of 5

Look-and-Say Sequence : Fifth Term



1 2 1 1



one "1" one "2" two "1"



1 1 1 2 2 1

5 of 5

— []

Now, you can easily guess the sixth term. There you go:

111221 is read off as "three 1s, two 2s, then one 1" or 312211.

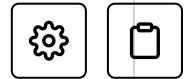
Implementation

Hopefully, by now, you understand the look-and-say sequence. Let's have a look at the implementation in Python below:

```
1 def next_number(s):
2     result = []
3     i = 0
4     while i < len(s):
5         count = 1
6         while i + 1 < len(s) and s[i] == s[i+1]:
7             i += 1
8             count += 1
9         result.append(str(count) + s[i])
```



```
10         i += 1
11     return ''.join(result)
```



next_number(s)

Explanation

In the `next_number` function, `result` is initialized to an empty list on **line 2**. `i` is set to `0` in the next line so that we can traverse the string, `s`, from the first character in the `while` loop on **line 4** which runs as long as `i` is less than the length of `s`. On **line 5**, in the outer `while` loop, `count` is set to `1` before the execution proceeds to the inner `while` loop. The inner `while` loop on **line 6** runs until the value of `i` does not exceed the length of `s` and the current and next character of `s` indicated by `i` are the same. So essentially, in the inner `while` loop, we are keeping a count of consecutive similar characters as we increment `i` and `count` by `1` in each iteration of the inner `while` loop (**lines 7-8**).

As soon as the inner `while` loop terminates due to change of character in the string, or the code has reached the end of the string, the execution jumps to **line 9**. On **line 9**, `count` is converted to a string and `s[i]`, which is the number we have the `count` for, is concatenated and appended to `result`. `i` is incremented on **line 10** to iterate to the next character in the next iteration of the outer `while` loop where `count` resets to `1` once again to count for the next number.

After `s` is traversed entirely, all the elements in `result` are concatenated using the `join` function, which joins all the elements in `result` without any space as specified by the `' '` separator. Then, a single string is returned from the function `next_number`.

Let's visualize the execution with the help of the slides below:



Look-and-Say Sequence

```
def next_number(s):  
    → result = []  
    i = 0  
    while i < len(s):  
        count = 1  
        while i + 1 < len(s) and s[i] == s[i+1]:  
            i += 1  
            count += 1  
        result.append(str(count) + s[i])  
        i += 1  
    return ''.join(result)
```

s = 11

result = []

1 of 12

Look-and-Say Sequence

```
def next_number(s):  
    result = []  
    → i = 0  
    while i < len(s):  
        count = 1  
        while i + 1 < len(s) and s[i] == s[i+1]:  
            i += 1  
            count += 1  
        result.append(str(count) + s[i])  
        i += 1  
    return ''.join(result)
```

s = 11

result = []

i = 0

2 of 12



Look-and-Say Sequence

```
def next_number(s):  
    result = []  
    i = 0  
    → while i < len(s):  
        count = 1  
        while i + 1 < len(s) and s[i] == s[i+1]:  
            i += 1  
            count += 1  
        result.append(str(count) + s[i])  
        i += 1  
    return ''.join(result)
```

s = 11

result = []

i = 0

i < len(s) ==> 0 < 2 ==> True

3 of 12

Look-and-Say Sequence

```
def next_number(s):  
    result = []  
    i = 0  
    while i < len(s):  
    → count = 1  
        while i + 1 < len(s) and s[i] == s[i+1]:  
            i += 1  
            count += 1  
        result.append(str(count) + s[i])  
        i += 1  
    return ''.join(result)
```

s = 11

result = []

i = 0

count = 1

4 of 12



Look-and-Say Sequence

```
def next_number(s):
    result = []
    i = 0
    while i < len(s):
        count = 1
        ➡ while i + 1 < len(s) and s[i] == s[i+1]:
            i += 1
            count += 1
        result.append(str(count) + s[i])
        i += 1
    return ''.join(result)
```

$(i + 1 < \text{len}(s) \text{ and } s[i] == s[i+1]) \Rightarrow (1 < 2 \text{ and } s[0] == s[1])$
 $\Rightarrow \text{True and } 1 == 1$
 $\Rightarrow \text{True and True} \Rightarrow \text{True}$

s = 11

result = []

i = 0

count = 1

5 of 12

Look-and-Say Sequence

```
def next_number(s):
    result = []
    i = 0
    while i < len(s):
        count = 1
        while i + 1 < len(s) and s[i] == s[i+1]:
            ➡ i += 1
            count += 1
        result.append(str(count) + s[i])
        i += 1
    return ''.join(result)
```

s = 11

result = []

i = 1

count = 1

6 of 12



Look-and-Say Sequence

```
def next_number(s):
    result = []
    i = 0
    while i < len(s):
        count = 1
        while i + 1 < len(s) and s[i] == s[i+1]:
            i += 1
            ➡ count += 1
        result.append(str(count) + s[i])
        i += 1
    return ''.join(result)
```

s = 11

result = []

i = 1

count = 2

7 of 12

Look-and-Say Sequence

```
def next_number(s):
    result = []
    i = 0
    while i < len(s):
        count = 1
        ➡ while i + 1 < len(s) and s[i] == s[i+1]:
            i += 1
            count += 1
        result.append(str(count) + s[i])
        i += 1
    return ''.join(result)
```

s = 11

result = []

i = 1

count = 2

(i + 1 < len(s) and s[i] == s[i+1]) ==> (2 < 2 and s[1] == s[2])
 ==> False and (Not evaluated)
 ==> False

8 of 12



Look-and-Say Sequence

```
def next_number(s):  
    result = []  
    i = 0  
    while i < len(s):  
        count = 1  
        while i + 1 < len(s) and s[i] == s[i+1]:  
            i += 1  
            count += 1  
        ➡ result.append(str(count) + s[i])  
        i += 1  
    return ''.join(result)
```

s = 11

result = ["21"]

i = 1

count = 2

9 of 12

Look-and-Say Sequence

```
def next_number(s):  
    result = []  
    i = 0  
    while i < len(s):  
        count = 1  
        while i + 1 < len(s) and s[i] == s[i+1]:  
            i += 1  
            count += 1  
        result.append(str(count) + s[i])  
        ➡ i += 1  
    return ''.join(result)
```

s = 11

result = ["21"]

i = 2

count = 2

10 of 12



Look-and-Say Sequence

```
def next_number(s):  
    result = []  
    i = 0  
    → while i < len(s):  
        count = 1  
        while i + 1 < len(s) and s[i] == s[i+1]:  
            i += 1  
            count += 1  
        result.append(str(count) + s[i])  
        i += 1  
    return ''.join(result)  
  
i < len(s) ==> 2 < 2 ==> False
```

s = 11
result = ["21"]
i = 2
count = 2

11 of 12

Look-and-Say Sequence

```
def next_number(s):  
    result = []  
    i = 0  
    while i < len(s):  
        count = 1  
        while i + 1 < len(s) and s[i] == s[i+1]:  
            i += 1  
            count += 1  
        result.append(str(count) + s[i])  
        i += 1  
    → return ''.join(result)  
  
"21" is returned from the function.
```

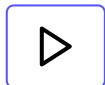
s = 11
result = ["21"]
i = 2
count = 2

12 of 12

— []

In the code below, by using a for loop on **line 16**, we generate the next term of the look-and-say sequence from the previous term and print out a total of four terms.

```
1 def next_number(s):
2     result = []
3     i = 0
4     while i < len(s):
5         count = 1
6         while i + 1 < len(s) and s[i] == s[i+1]:
7             i += 1
8             count += 1
9         result.append(str(count) + s[i])
10        i += 1
11    return ''.join(result)
12
13 s := "1"
14 print(s)
15 n := 4
16 for i in range(n-1):
17     s := next_number(s)
18     print(s)
```



Output

0.74s

```
1
11
21
1211
```

I hope you were able to understand the string processing regarding the look-and-say sequence that we performed in this lesson. Stay tuned for more problems regarding string processing in the next few lessons.

[← Back](#)[Next →](#)

Quiz

Spreadsheet Encoding

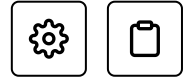


Report an
Issue



Ask a Question

(https://discuss.educative.io/tag/look-and-say-sequence__string-processing__data-structures-and-algorithms-in-python)



Spreadsheet Encoding

In this lesson, you will learn how to convert the column IDs in a spreadsheet into an integer in Python.

We'll cover the following



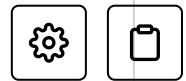
- Implementation
- Explanation

In this lesson, we will be considering how to solve the problem of implementing a function that converts a spreadsheet column ID (i.e., “A”, “B”, “C”, ..., “Z”, “AA”, etc.) to the corresponding integer. For example, “A” equals 1 because it represents the first column, while “AA” equals 27 because it represents the 27th column.

We will cover how to solve this problem algorithmically, and then code a solution to this question in Python.

The key insight is to think of the column IDs as base 26 integers.

In the illustration below, you can see how we represent “314” as a base 10 number.



314



$$3*100 + 1*10 + 4*1$$



$$3*10^2 + 1*10^1 + 4*10^0$$

From the illustration above, you can easily deduce the general formula. We start from the left-most digit and multiply it with the base raised to the power of $(n - 1)$ where n is the number of digits in that number. We continue with the same procedure with the other digits but keep subtracting one from the power of the base, so the power of the base will be 0 for the last digit. Finally, the sum of all the products is equivalent to the original number.

Now, let's go ahead and represent the numbers of base 26 according to the formula above. Here we go:



A = 1 , B = 2, ..., Z = 26

AA

$$A * 26^1 + A * 26^0$$

$$1 * 26^1 + 1 * 26^0 = 27$$

Implementation

At this point, the base representation of a number will be pretty clear to you. Let's make use of it in the implementation in Python, but before we jump to the implementation, let's get introduced to `ord` function in Python:

```
1 print(ord('A'))  
2 print(ord('B'))  
3 print(ord('Z'))
```



Output

0.72s

```
65  
66  
90
```




`ord()` returns an integer which represents the Unicode code point of the Unicode character passed into the function. For the solution to work, we need to make sure of the following:

$A = 1, B = 2, \dots, Z = 26$

as we deal with base 26 numbers.

To make A represent 1, B represent 2, and so on, we need to determine the relative difference from the result given by the `ord` function and from the representation we require for base 26 system. Now we know that `ord('A')` equals 65. So if we find the Unicode code point using `ord()` of a character, subtract `ord('A')` from it, and then add 1 to it, we'll get the representation we want for the base 26 system. For instance, check out the code below for the characters that have been converted with this methodology:

```
1 print(ord('A')- ord('A')+ 1)
2 print(ord('B')- ord('A')+ 1)
3 print(ord('C')- ord('A')+ 1)
4 print(ord('Z')- ord('A')+ 1)
```



Output

0.85s

1
2
3
26

Now let's discuss the actual implementation:

```
1 def spreadsheet_encode_column(col_str):
2     num = 0
3     count = len(col_str)-1
4     for s in col_str:
5         num += 26**count * (ord(s) - ord('A') + 1)
6         count -= 1
7     return num
8
9
10 print(spreadsheet_encode_column("ZZ"))
```



Output

0.76s

702

spreadsheet_encode_column(col_str)

Explanation

num is set to 0 initially on **line 2** and returned from the function on **line 7**. count is initialized to $\text{len}(\text{col_str}) - 1$ where col_str is the string representing the columnID that we have to convert into a number. count will help us in determining the power of the base as we convert the columnID into a corresponding integer. On **line 4**, s will represent each character of col_str in each iteration of the for loop. Let's jump to **line 5** where 26 (base) is raised to the power of count where count is $\text{len}(\text{col_str}) - 1$ as indicated in the formula. The result from the power operation (**) is multiplied with the equivalent of s in base 26 system which we find using the ord function. The final answer as a result of these calculations is added to the value of the existing num on **line 5**. On **line 6**, count is decremented by 1 as the power of the base decreases by 1 for the

digits from left to the right. After the calculations have been done for every character in `s`, the `for` loop terminates, and the value of `num` is returned from the function.



That's all we have for this lesson. Hope you are having a great learning experience. Let's move on to discussing another problem in the next lesson.

[← Back](#)[Next →](#)[Look-and-Say Sequence](#)[Is Palindrome](#)[Mark as Completed](#)[Report an Issue](#)[Ask a Question](#)

(https://discuss.educative.io/tag/spreadsheet-encoding__string-processing__data-structures-and-algorithms-in-python)



Is Palindrome

In this lesson, you will learn how to determine if a string is a palindrome or not.

We'll cover the following



- Implementation
 - Solution 1
 - Solution 2: The Preferred Solution

In this lesson, we will be considering how to test whether a string is a palindrome in Python. At first, we'll come up with a concise solution that takes extra space, but we'll eventually code a solution that takes a linear amount of time and a constant amount of space.

A palindrome is a word, number, phrase, or any other sequence of characters that reads the same forward as it does backward.

Here are some examples of a palindrome:



Examples of palindromes

Was it a cat I saw

Never odd or even

radar

Live on time, emit no evil

Implementation

Let's get started with the implementations.

Solution 1

```
1 s = "Was it a cat I saw?"
2
3 #Solution uses extra space proportional to size of s
4 s = ''.join([i for i in s if i.isalnum()]).replace(" ", "")
5 print(s == s[::-1])
```



Output

0.85s

True

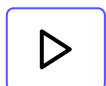


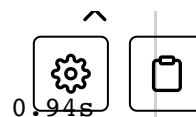
Let's discuss the above one-line solution in parts. So the `[i for i in s if i.isalnum()]` statement returns a list of all the characters which are alphanumeric. This helps us get rid of the question marks, the spaces and every character which is not alphanumeric. The `join` function in the same line then joins the elements from this list into a string which is turned into lowercase via `lower` function.

Finally, on **line 5**, `s` after modifications on **line 4**, is compared with its reversed form. If they are both equal, `True` is printed on the console otherwise `False` will be printed on to the console. The above solution is very concise, but as it requires extra space, we prefer another solution which will solve the problem in linear time. Let's go!

Solution 2: The Preferred Solution

```
1 def is_palindrome(s):
2     i = 0
3     j = len(s) - 1
4
5     while i < j:
6         while not s[i].isalnum() and i < j:
7             i += 1
8         while not s[j].isalnum() and i < j:
9             j -= 1
10
11        if s[i].lower() != s[j].lower():
12            return False
13        i += 1
14        j -= 1
15    return True
16
17 s = "Was it a cat I saw?"
18 print(is_palindrome(s))
```





Output

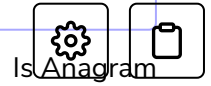
True

is_palindrome(s)

`i` and `j` are two iterators which point to the first and the last character of `s` (**lines 2-3**). Next, we have a `while` loop on **line 5** which runs as long as `i` is less than `j`. `i` is supposed to go forward from the beginning of `s` while `j` has to go backward from the end of `s`. So, on **line 6** and on **line 8**, we have two `while` loops which increment or decrement the iterators accordingly. So `i` is incremented on **line 7** if `s[i]` is not an alphanumeric and `i` is less than `j`. Similarly, `j` is decremented on **line 9** if `s[j]` is not alphanumeric and `i` is less than `j`. This will help us ignore all the characters in `s` which are not alphanumeric.

On **line 11**, we check for the palindrome property by comparing `s[i]` and `s[j]`. They have to be the same for `s` to be a palindrome. However, if `s[i]` and `s[j]` do not match, `False` is returned from the function on **line 12** to indicate an instance where the character pointed to by `i` is not the same as the character pointed to by `j`. If `s[i]` equals `s[j]`, the execution proceeds to **lines 13-14** where `i` is incremented and `j` is decremented to move to the next characters. If the condition on **line 11** stays `False` for all possible iterations of the `while` loop, `True` is returned on **line 15** to indicate that `s` is indeed a palindrome.

That sums up everything! In this lesson, you learned how to check if a string is a palindrome or not. In the next lesson, we'll explore how to find if a string is an anagram or not. See you there!

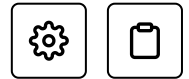
 **Back****Next** 

Spreadsheet Encoding

☒ Mark as CompletedReport an
Issue

Ask a Question

(https://discuss.educative.io/tag/is-palindrome__string-processing__data-structures-and-algorithms-in-python)



Is Anagram

In this lesson, you will learn how to determine if a string is an anagram of another string or not.

We'll cover the following



- Implementation
 - Solution 1
 - Solution 2: The Preferred Solution

In this lesson, we will determine whether two strings are anagrams of each other.

Simply put, an anagram is when two strings can be written using the same letters.

Examples:

```
"rail safety" = "fairy tales"  
"roast beef" = "eat for BSE"
```

It sometimes changes a proper noun or personal name into a sentence:

```
"William Shakespeare" = "I am a weakish speller"  
"Madam Curie" = "Radium came"
```

We will write two solutions to this problem. The first one will be concise, while the second one will be more robust and efficient. Both of these approaches involve normalizing the input string, which means converting

them into lowercase and removing all the characters which are not alphanumeric.



Let's get started with the first approach.

Implementation

Solution 1

```
1 s1 = "fairy tales"
2 s2 = "rail safety"
3
4 s1 = s1.replace(" ", "").lower()
5 s2 = s2.replace(" ", "").lower()
6
7 #Requires n·log·n·time·(since·any·comparison·
8 #based·sorting·algorithm·requires·at·least·
9 #n·log·n·time·to·sort).
10 print(sorted(s1) == sorted(s2))
```



Output

0.81s

True

On **lines 4-5**, `s1` and `s2` are normalized using `replace()` and `lower()` functions. `replace(" ", "")` replaces all the spaces with an empty string so that we are only left with alphabets or numbers. `lower()` converts all the alphabets to lowercase. On **line 10**, the two strings are sorted after the normalization and then compared. If `s1` and `s2` are anagrams of each

other, both the lists returned from `sorted` will be the same and `True` will be printed onto the console. On the other hand, if `sorted(s1)` does not equal `sorted(s2)`, `False` will be printed onto the console.



You may notice that the solution above is very concise. However, the time complexity for the above algorithm will be $O(n \log n)$ as sorting takes $O(n \log n)$. Therefore, we prefer the second approach, which is more efficient than the sorting approach.

Solution 2: The Preferred Solution

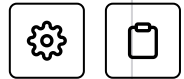
In this solution, we make use of a Python dictionary. First, we store the characters of one string in a dictionary, keeping track of the count of individual characters. Then we compare the count of each character with the characters of the other string. If they counterbalance each other, we'll declare the two strings as an anagram of each other. This solution is of linear time complexity which is an improvement on $O(n \log n)$.

Have a look at the code below to see how we have implemented the approach mentioned above:

```
1 def is_anagram(s1, s2):
2     ht = dict()
3
4     if len(s1) != len(s2):
5         return False
6
7     for i in s1:
8         if i in ht:
9             ht[i] += 1
10        else:
11            ht[i] = 1
12    for i in s2:
13        if i in ht:
14            ht[i] -= 1
15        else:
16            ht[i] = 1
17    for i in ht:
18        if ht[i] != 0:
```



```
19         return False
20     return True
21
22     s1 = "fairy tales"
23     s2 = "rail safety"
24     ## normalizing the strings
25     s1 = s1.replace(" ", "").lower()
26     s2 = s2.replace(" ", "").lower()
27
28     print(is_anagram(s1, s2))
```



Output

1.05s

True

On **line 2**, `ht` is initialized to a Python dictionary. For two strings to be anagrams of each other, they must be of the same length. Therefore, on **line 4**, we compare the length of the two strings. If they are not equal, `False` is returned on **line 5** before any more computation. If `s1` and `s2` have the same number of characters, the execution jumps to the `for` loop on **line 7**.

This `for` loop iterates over each character of `s1` and checks if it is present in `ht`. If it's not, we store the character `i` as a key in `ht` while setting its value to `1` on **line 11**. If `i` is present in `ht`, then the code on **line 9** executes which increments the value against `i` in `ht` by `1`. Now that we have the count of characters of `s1` stored in `ht`, the code proceeds to the `for` loop on **line 12**. The `for` loop on **line 12** is exactly the same as the previous `for` loop but only with a minor difference. This `for` loop iterates over `s2` and if `i` is present in `ht` as a key, instead of incrementing its value, it decrements its value by `1`.

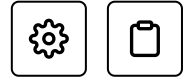
At this point, if `s1` and `s2` are anagrams of each other, all the keys in `ht` will have `0` as their values. This is because the count of each character in `s2` will cancel out the count of each character in `s1` because they have the same number of each type of character. So, on **line 17**, as we iterate over every key in `ht`, if `ht[i]` does not equal `0`, `False` is returned from the function to indicate that `s1` and `s2` are not anagrams of each other. If the condition on **line 18** never evaluates to `True`, `True` is returned from the function on **line 20** to declare `s1` and `s2` as anagrams.

Also, note that `s1` and `s2` are normalized on **lines 25-26** to remove the spaces which might result in uneven lengths even if `s1` and `s2` are anagrams of each other.

That is all we have up for this lesson. In the next lesson, we'll study "Is Palindrome Permutation". Stay tuned!

[← Back](#)[Next →](#)[Is Palindrome](#)[Is Palindrome Permutation](#)☒ Mark as Completed[Report an Issue](#)[Ask a Question](#)

(https://discuss.educative.io/tag/is-anagram__string-processing__data-structures-and-algorithms-in-python)



Is Palindrome Permutation

In this lesson, you will learn how to find if a string is a palindrome permutation or not.

We'll cover the following



- Implementation
- Explanation

In this lesson, we will consider how to determine if a string is a palindrome permutation.

Specifically:

Given a string, write a function to check if it is a permutation of a palindrome.

Recall:

Palindrome: A word or phrase that is the same forwards and backward.

Permutation: A rearrangement of letters.

The palindrome does not need to be limited to just dictionary words.

Before jumping straight to the code, let's think about how we can approach this problem. Here is an example of a palindrome:



Example of a Palindrome

taco cat

Look closely and see if you can identify a pattern in the above palindrome.

1 of 2

Example of a Palindrome

t a c o c a t

Every character has a duplicate in this odd-length string except the one character at the middle.

2 of 2

From the slides above, the idea we get is that a string that has an even length must have all even counts of characters, while strings that have an odd length must have exactly one character with an odd count. An even-lengthed string can't have an odd number of exactly one character; otherwise, it wouldn't be even. This is true since an odd number plus any set of even numbers will yield an odd number.



Alternatively, an odd-lengthed string can't have all characters with even counts, the sum of any number of even numbers is even. We can, therefore, say that it's sufficient to be a permutation of a palindrome, that is a string can have no more than one character that is odd.

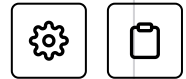
Implementation

Let's make use of the observation discussed above regarding the palindrome property to implement the code in Python. We use a Python dictionary to solve the problem in the code below:

```
1 def is_palin_perm(input_str):
2     input_str = input_str.replace(" ", "")
3     input_str = input_str.lower()
4
5     d = dict()
6
7     for i in input_str:
8         if i in d:
9             d[i] += 1
10        else:
11            d[i] = 1
12
13    odd_count = 0
14    for k, v in d.items():
15        if v % 2 != 0 and odd_count == 0:
16            odd_count += 1
17        elif v % 2 != 0 and odd_count != 0:
18            return False
19    return True
20
21 palin_perm = "Tact Coa"
```




```
22 not_palin_perm = "This is not a palindrome permuta
23
24 print(is_palin_perm(palin_perm))
25 print(is_palin_perm(not_palin_perm))
```



Output

1.02s

True
False

is_palin_perm()

Explanation

Lines 2-3 are covering the normalization process which we also discussed in the previous lesson. Once all the spaces are removed from `input_str` and the characters in `input_str` are converted to lowercase, a dictionary named `d` is defined on **line 5**.

On **line 7**, by using a `for` loop, all the characters in `input_str` are stored as keys in `d` with their values equal to their number of occurrences in `input_str`.

`odd_count` is initialized to `0` on **line 13**. Next, we loop over `d` using the `for` loop on **line 14** where `k` is the key of each item, and `v` is the value for that key. On **line 15**, we check if `v` is an odd number and `odd_count` equals `0`. If for some `k`, `v` is odd and `odd_count` is `0`, we increment `odd_count` by `1` on **line 6**. In this way, we are recording for an instance if we have encountered the middle element of an odd-length string. However, on **line 17**, if `v` is an odd number, but `odd_count` equals something other than `0`, it

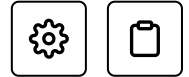
implies that there is more than one character which has an odd number of occurrences in `input_str`. Therefore, `False` is returned on **line 18** to indicate that the condition on **line 17** is `True` and `input_str` is not a permutation of a palindrome. On the other hand, if all goes well and the condition on **line 17** never evaluates to `True`, `True` is returned on **line 19** to indicate a positive response.



I hope this explanation clarifies the implementation. In the next lesson, we will discuss the implementation to solve the “Check Permutation” problem. Happy learning!

[← Back](#)[Next →](#)[Is Anagram](#)[Check Permutation](#)[Mark as Completed](#)[Report an Issue](#)[Ask a Question](#)

(https://discuss.educative.io/tag/is-palindrome-permutation__string-processing__data-structures-and-algorithms-in-python)



Check Permutation

In this lesson, you will learn how to check if a string is a permutation of another string in Python.

We'll cover the following



- Solution 1
 - Implementation
 - Explanation
- Solution 2
 - Implementation
 - Explanation

In this lesson, we will consider how to determine if a given string is a permutation of another string.

Specifically, we want to solve the following problem:

Given two strings, write a function to determine if one is a permutation of the other.

Here is an example of strings that are permutations of each other:

```
is_permutation_1 = "google"  
is_permutation_2 = "ooggle"
```

The strings below are not permutations of each other.

```
not_permutation_1 = "not"  
not_permutation_2 = "top"
```



We will solve this problem in Python and analyze the time and space complexity of our approach.

Let's begin!

Solution 1

Implementation

A permutation of a string will have the same number of each type of character as that string as it is just a rearrangement of the letters.

Let's have a look at the code below where we make use of this property:

```
1 # Approach 1: Sorting  
2 # Time Complexity: O(n log n)  
3 # Space Complexity: O(1)  
4 def is_perm_1(str_1, str_2):  
5     ... str_1 = str_1.lower()  
6     ... str_2 = str_2.lower()  
7  
8     ... if len(str_1) != len(str_2):  
9         ... return False  
10  
11     ... str_1 = ''.join(sorted(str_1))  
12     ... str_2 = ''.join(sorted(str_2))  
13  
14     ... n = len(str_1)  
15  
16     ... for i in range(n):  
17         ... if str_1[i] != str_2[i]:  
18             ... return False  
19     ... return True
```





Explanation

First of all, both the strings `str_1` and `str_2` are normalized on **lines 5-6** as all the characters are converted to lowercase. On **line 8**, we check for a basic permutation property, i.e., the two strings that are permutations of each other have to be of the same length. Therefore, `False` is returned on **line 9** if the length of `str_1` does not equal length of `str_2`.

On **lines 11-12**, both the strings are sorted using the `sorted` function which returns a list. That list is again turned into a string using the `join` function. At this point, `str_1` and `str_2` will be two strings with all the characters sorted in them.

Now if `str_1` is a permutation of `str_2`, all the characters will be the same in the sorted version of both the strings. We'll check this using the `for` loop on **line 16** which runs from `i` equal `0` to `i` equal `n-1`. To traverse the strings, we make use of `n` which is set to `len(str_1)`. So, on **line 17**, we check for the equality of the characters on the same index. If, in any iteration, `str_1[i]` is not equal to `str_2[i]`, `False` is returned on **line 18**. Otherwise, if `False` is never returned from the function, `True` is returned on **line 19**.

As we are using sorting to check for permutations, the time complexity for this solution is $O(n \log n)$ while space complexity is $O(1)$ as there is no extra use of space.

Solution 2

As Solution 1 has $O(n \log n)$ complexity, we need something better. Solution 2 makes use of a hash table to reach a time complexity of $O(n)$.

Implementation

Let's find out how by having a look at the code below:



```
1 # Approach 2: Hash Table
2 # Time Complexity: O(n)
3 # Space Complexity: O(n)
4 def is_perm_2(str_1, str_2):
5     str_1 = str_1.lower()
6     str_2 = str_2.lower()
7
8     if len(str_1) != len(str_2):
9         return False
10
11     d = dict()
12
13     for i in str_1:
14         if i in d:
15             d[i] += 1
16         else:
17             d[i] = 1
18     for i in str_2:
19         if i in d:
20             d[i] -= 1
21         else:
22             return False
23
24     return all(value == 0 for value in d.values())
```

is_perm_2()

Explanation

The code from **line 5** to **line 9** is the same as in Solution 1. On **line 11**, `d` is initialized to a Python dictionary. Using a `for` loop on **line 13** where `i` equals a character in `str_1`, `i` is stored as a key in `d` with `1` as a value if it's not present in `d`. If it's present, its value is incremented by `1`. After this `for` loop, `d` will have the entire count of all the characters present in `str_1`. We'll make use of this in the `for` loop on **line 18** where `str_2` is

traversed. If any character in `str_2` is present as a key in `d`, its value is decremented by 1. If it's not already present in `d`, it is inserted into `d` as a key with value equal to 1.



Now, if `str_1` and `str_2` are permutations of each other, the two `for` loops will counterbalance each other and the values of all the keys in `d` should be 0. We find this out on **line 24** where we evaluate `value == 0` for all the keys in `d`. Using the `all` function, we combine the evaluation results from all the iterations and return it from the function. The `all()` function returns `True` if all items in an iterable are `True`. Otherwise, it returns `False`.






In the code widget below, you can execute both the functions on different test cases.

```

29     if len(str_1) != len(str_2):
30         return False
31
32     d = dict()
33
34     for i in str_1:
35         if i in d:
36             d[i] += 1
37         else:
38             d[i] = 1
39     for i in str_2:
40         if i in d:
41             d[i] -= 1
42         else:
43             return False
44
45     return all(value == 0 for value in d.values())
46
47 is_permutation_1 = "google"
48 is_permutation_2 = "ooggle"
49
50 not_permutation_1 = "not"
51 not_permutation_2 = "top"
52 print(is_perm_1(is_permutation_1, is_permutation_2))
53 print(is_perm_1(not_permutation_1, not_permutation_2))
54
55 print(is_perm_2(is_permutation_1, is_permutation_2))
56 print(is_perm_2(not_permutation_1, not_permutation_2))

```





Output

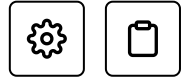
0.72s

True
False
True
False

Now that you are pretty familiar with the problems regarding string processing, it's test time! Brace yourselves for the upcoming challenge!

[← Back](#)[Next →](#)[Is Palindrome Permutation](#)[Exercise: Is Unique](#)[Mark as Completed](#)[Report an Issue](#)[Ask a Question](#)

(https://discuss.educative.io/tag/check-permutation__string-processing__data-structures-and-algorithms-in-python)



Exercise: Is Unique

Challenge yourself with an exercise in which you'll have to determine whether a string is unique or not.

We'll cover the following ^

- Problem
- Coding Time!

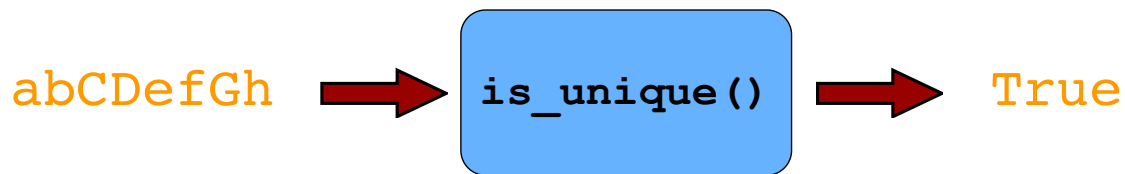
Problem

Your task is to implement an algorithm to determine if a string has all unique characters.

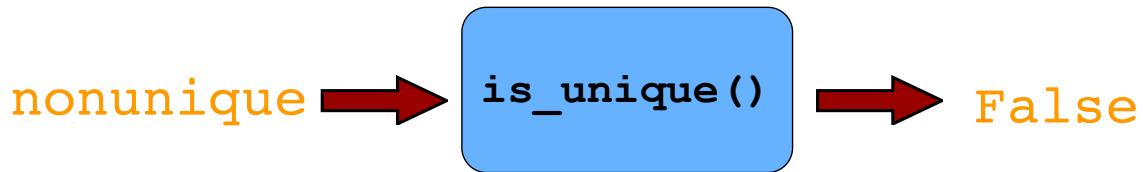
Assume that the input string will only contain alphabets or spaces.



Unique String



Non-unique String



Coding Time!

Your task is to return `True` or `False` from `is_unique` function, which will be based on whether or not `input_str` has all unique characters.


Good luck!

```
1 def is_unique(input_str):
2     input_str = input_str.replace(" ", "")
3     input_str = input_str.lower()
4     d = dict()
5     for c in input_str:
6         if c in d:
7             d[c] += 1
8         else:
9             d[c] = 1
10    return all(val == 1 for val in d.values())
```



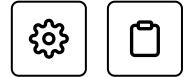
Show Results**Show Console**

0.88s

 **4 of 4 Tests Passed**

Result	Input	Expected Output	Actual Output	Reason
✓	<code>is_unique(abCedFghI)</code>	True	True	Succeeded
✓	<code>is_unique(I Am Not Unique)</code>	False	False	Succeeded
✓	<code>is_unique(heythere)</code>	False	False	Succeeded
✓	<code>is_unique(hi)</code>	True	True	Succeeded

[< Back](#)[Next >](#)



Solution Review: Is Unique

This lesson contains the solution review to determine whether a string contains all unique characters or not.

We'll cover the following



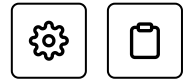
- Normalization
- Solution 1
 - Implementation
 - Explanation
- Solution 2
 - Implementation
 - Explanation
- Solution 3
 - Implementation
 - Explanation

In this lesson, we will consider how to determine if a string has all unique characters. One approach to this problem may be to use an additional data structure, like a hash table. We will also consider how one may solve this problem without the use of such a data structure.

So, we'll present a number of solutions to the problem posed in the previous challenge and give a rough time and space complexity analysis of each approach.

Let's get started!

Normalization



First of all, to process the strings, we need to normalize them. The normalization process is as follows:

```
1 def normalize_str(input_str):  
2     ...input_str = input_str.replace(".", "")  
3     ...return input_str.lower()
```



Solution 1

Now let's discuss the actual implementation. Solution 1 uses a Python dictionary to solve the problem in linear time complexity, but because of the additional data structure, the space complexity is also linear.

Implementation

Below is the implementation of Solution 1 in Python:

```
1 def is_unique_1(input_str):  
2     ...d = dict()  
3     ...for i in input_str:  
4         ...if i in d:  
5             ...return False  
6         ...else:  
7             ...d[i] = 1  
8     ...return True
```



Explanation

`d` is initialized to a Python dictionary on **line 2**. Next, there is a `for` loop on **line 3** which iterates `input_str` character by character. In this `for` loop, the condition on **line 4** checks if `i`, i.e., the current character, is present in `d` or

not. If it's not present in `d`, the execution jumps to **line 7** where it is then inserted into `d` as a key with `1` as its value. This is how we record the first instance of any character. However, if `d` is already present in `d`, it means that we encounter it on its second occurrence, which implies that it is not a unique character. Therefore, `False` is returned from the function in case `i` is present in `d`.



If `False` is never returned from the function and the `for` loop terminates, `True` is returned on **line 8**.

I hope everything's been clear up until now!

Solution 2

Now, solution 2 is very concise and straightforward. In this solution, we make use of `set()`. Let's find out how by having a look at the implementation in Python.

Implementation

```
1 def is_unique_2(input_str):  
2     ...return len(set(input_str)) == len(input_str)
```



Explanation

Did you check out how simple the solution looks? Let's discuss it.

`set(input_str)` converts `input_str` into a set by removing all the duplicates. Now if the length of that set is equal to the length of `input_str`, it implies that `input_str` did not have any duplicates and has all unique characters. Yes, it is as simple as that!



As we have no idea about how the function `set()` works internally, we cannot be sure about the time and space complexity. However, my understanding of the built-in `set` function is that it is going to take linear time to process all of the elements to determine the set of the list.

Solution 3

It's time for Solution 3. Let's jump to the implementation in Python!

Implementation

```
1 def is_unique_3(input_str):
2     ...alpha = "abcdefghijklmnopqrstuvwxyz"
3     ...for i in input_str:
4         ...if i in alpha:
5             ...alpha = alpha.replace(i, "")
6         ...else:
7             ...return False
8     ...return True
```



Explanation

Solution 3 is pretty straightforward. `alpha` is a string defined on **line 2** which contains all 26 letters in lowercase. The `for` loop on **line 3** traverses all the characters in `input_str`. If a character of `input_str`, i.e., `i`, is present in `alpha`, we replace it with an empty string and update `alpha` accordingly on **line 5**. Now as we keep removing `i` in each iteration from `alpha`, if in any iteration we encounter an `i` which is not in `alpha`, it means that it was removed in the previous iterations. The execution jumps to **line 7** and `False` is returned to signal for a duplicate character.

However, if the condition on **line 4** is never `False` in any iteration of the `for` loop, `True` is returned from the function on **line 8** to indicate that `input_str` has all unique characters.

In the code widget below, you can find and execute all three functions that we have discussed above.



```
26         return False
27     return True
28
29 unique_str = "AbCDefG"
30 non_unique_str = "non Unique STR"
31
32 unique_str = normalize_str(unique_str)
33 non_unique_str = normalize_str(non_unique_str)
34 print("Unique String")
35 print(unique_str)
36 print("Non-Unique String")
37 print(non_unique_str, "\n")
38
39 print("Solution 1 where input string is unique string")
40 print(is_unique_1(unique_str))
41 print("Solution 1 where input string is non-unique string")
42 print(is_unique_1(non_unique_str), "\n")
43
44
45 print("Solution 2 where input string is unique string")
46 print(is_unique_2(unique_str))
47 print("Solution 2 where input string is non-unique string")
48 print(is_unique_2(non_unique_str), "\n")
49
50 print("Solution 3 where input string is unique string")
51 print(is_unique_3(unique_str))
52 print("Solution 3 where input string is non-unique string")
53 print(is_unique_3(non_unique_str))
```



Output

0.87s

Unique String

abcdefg

Non-Unique String

nonuniquestr

Solution 1 where input string is unique string

True

Solution 1 where input string is non-unique string



In the next lesson, we will learn how to convert an integer to a string in Python. See you there!

← Back

Next →

Exercise: Is Unique

Integer to String



Mark as Completed

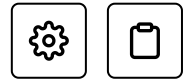


Report an
Issue



Ask a Question

(https://discuss.educative.io/tag/solution-review-is-unique__string-processing__data-structures-and-algorithms-in-python)



Integer to String

In this lesson, you will learn how to convert an integer to a string in Python.

We'll cover the following



- Implementation
- Explanation

In this lesson, we will solve the following problem:

You are given some integer as input, (i.e. ... -3, -2, -1, 0, 1, 2, 3 ...) and you have to convert the integer you are given to a string. Examples:

```
Input: 123
Output: "123"
Input: -123
Output: "-123"
```

Note that you cannot make use of the built-in `str` function:

```
1 print(str(123))
2 print(type(str(123)))
```



Output

0.96s

```
123
<class 'str'>
```



Before diving into the implementation of the solution, we need to get familiar with the following functions:

1. `ord()`
2. `chr()`

You might be able to recall `ord()` from one of the previous lessons. `ord()` returns an integer which represents the Unicode code point of the Unicode character passed into the function. On the other hand, `chr()` is the exact opposite of `ord()`. `chr()` returns a character from an integer that represents the Unicode code point of that character.

```
1  ## Prints 48 which is the Unicode code point of the character '0'
2  print(ord('0'))
3  ## Prints the character '0' as 48 is Unicode code point of '0'
4  print(chr(ord('0')))
5  ## Prints 49
6  print(ord('0')+1)
7  ## Prints 49 which is Unicode code point of the character '1'
8  print(ord('1'))
9  ## Prints the character '2' as 50 is Unicode code point of '2'
10 ## ord('0')+2 == 48+2 == 50
11 print(chr(ord('0')+2))
12 ## Prints the character '3' as 51 is Unicode code point of '3'
13 ## ord('0')+3 == 48+3 == 51
14 print(chr(ord('0')+3))
```



Output

0.83s

48
0
49

49
2
3



From the above coding example, you can observe the following pattern:

```
ord('0') = 48
ord('1') = ord('0') + 1 = 48 + 1 = 49
ord('2') = ord('0') + 2 = 48 + 2 = 50

chr(ord('0')) = chr(48) = '0'
chr(ord('0') + 1) = chr(48 + 1) = chr(49) = '1'
chr(ord('0') + 2) = chr(48 + 2) = chr(50) = '2'
```

At this point, you must be familiar with the workings of `ord` and `chr` functions. Let's discuss the actual implementation now.

Implementation

```
1 def int_to_str(input_int):
2
3     if input_int < 0:
4         is_negative = True
5         input_int *= -1
6     else:
7         is_negative = False
8
9     output_str = []
10
11     if input_int == 0:
12         output_str.append('0')
13     else:
14         while input_int > 0:
15             output_str.append(chr(ord('0') + input_int % 10))
16             input_int //= 10
17         output_str = output_str[::-1]
18
19     output_str = ''.join(output_str)
20
21     if is_negative:
```



```
22         return '-' + output_str
23     else:
24         return output_str
25
26 input_int = 123
27 print(input_int)
28 print(type(input_int))
```



Output

1.11s

```
123
<class 'int'>
123
<class 'str'>
```

`int_to_str(input_int)`

Explanation

From **line 3** to **line 7**, we determine whether `input_int` is a negative or a positive integer:

```
if input_int < 0:
    is_negative = True
    input_int *= -1
else:
    is_negative = False
```

If `input_int` is less than 0, `is_negative` is set to `True` on **line 4** and is also converted to a positive integer by multiplication with `-1` on **line 5**. On the other hand, if `input_int` is a positive integer, `is_negative` is set to `False` on **line 7**.

Once the sign of `input_int` is determined, the execution jumps to **line 9**



Here is the code from **lines 9-17**:

```
output_str = []

if input_int == 0:
    output_str.append('0')
else:
    while input_int > 0:
        output_str.append(chr(ord('0') + input_int % 10))
        input_int //= 10
    output_str = output_str[::-1]
```

`output_str` is initialized to an empty list on **line 9**. If `input_int` is 0, the result is straightforward, i.e., '0'. But if the number is greater than 0, then the while loop on **line 14** is where the actual action happens. The last digit of `input_int` is extracted using the `%` operator on **line 15**.

For example,

```
12 % 10 = 2
17 % 10 = 7
10 % 10 = 0
```

You notice that by taking the modulus of a number with 10, we always get the last digit of that number. The last digit is converted into a character by `chr()` and `ord()` functions. So `ord('0') + input_int % 10` gives the Unicode code point of the character that we want which when passed to `chr()` function returns a character on **line 15**. That character is appended to `output_str` in the same line. On **line 16**, as we have already dealt with the last digit, we remove it from `input_int` using the `//` operator. The while loop repeats the code on **lines 15-16** until `input_int` becomes less than or equal to 0.

On **line 17**, `output_str` reverses the positions of its elements so that the last digit is on the last index of `output_str` instead of being on the first one. That brings us to the code on **lines 19- 24**:

```
output_str = ''.join(output_str)

    if is_negative:
        return '-' + output_str
    else:
        return output_str
```

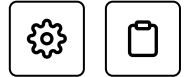
On **line 19**, the elements in `output_str` are joined together using the `join` function. The `''` acts as a separator which translates to the fact that there will be nothing between the elements as `''` is just an empty string. Now that we have `input_int` converted to an integer, the only thing left for us is to deal with its sign. If `is_negative` is `True`, we add a `-` before `output_str` and return it on **line 22**, otherwise `output_str` is returned without any further modification on **line 24**.

You can confirm the type of outputs from the built-in `type` function in Python.

So that was how we convert an integer to a string in Python. Interesting, right? Now get ready for a challenge in the next lesson where you'll have to convert a string into an integer — excited, right? See you there!

[← Back](#)[Next →](#)[Solution Review: Is Unique](#)[Exercise: String to Integer](#)☒ Mark as Completed[Report an Issue](#)[Ask a Question](#)https://discuss.educative.io/tag/integer-to-string__string-processing__data-structures-and-algorithms-in-python





Exercise: String to Integer

Challenge yourself with an exercise in which you'll have to convert a string into an integer.

We'll cover the following ^

- Problem
- Coding Time

Problem

In this exercise, you are given some numeric string as input. You have to convert the string you are given to an integer.

Do not make use of the built-in `int` function.

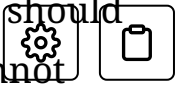
Example:

```
"123" -> 123  
"-12332" -> -12332  
"554" -> 554
```

 Show Hint

Coding Time

Your task is to return an integer from the `str_to_int` function which should equal the number represented by `input_str`. Remember that you cannot make use of the built-in `int` function. Solving this problem without built-in functionality will help you in interviews.



Good luck!

```
1 def str_to_int(input_str):
2     output = 0
3     if input_str[0] == "-":
4         is_neg = True
5         input_str = input_str[1:]
6     else:
7         is_neg = False
8     for i in input_str:
9         output = output*10 + (ord(i)-ord("0"))
10    if is_neg:
11        output *= -1
12    return output
```



Show Results

Show Console



0.81s

4 of 4 Tests Passed

Result	Input	Expected Output	Actual Output	Reason
✓	<code>str_to_int(0)</code>	0	0	Succeeded
✓	<code>str_to_int(-1293)</code>	-1293	-1293	Succeeded
✓	<code>str_to_int(529)</code>	529	529	Succeeded
✓	<code>str_to_int(-999)</code>	-999	-999	Succeeded

 Back

Integer to String

Solution Review: String to Integer

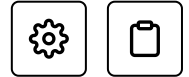


Mark as Completed

Report an
Issue

Ask a Question

(https://discuss.educative.io/tag/exercise-string-to-integer__string-processing__data-structures-and-algorithms-in-python)



Solution Review: String to Integer

This lesson contains the solution review to convert a string into an integer in Python.

We'll cover the following



- Implementation
- Explanation

In this lesson, we will review the solution to the challenge in the last lesson. The problem is as follows:

Given some numeric string as input, convert the string to an integer.

You already know that we are not allowed to use the built-in `int()` function. Now to convert a string into an integer, we can make use of the following arithmetic:



$$\begin{array}{c} 123 \\ \downarrow \\ 1*10^2 + 2*10^1 + 3*10^0 \\ \downarrow \\ 100 + 20 + 3 = 123 \end{array}$$

As you can see, by multiplying each digit with the appropriate base power, we can get the original integer back. Let's implement this in Python.

Implementation

```
5     if input_str[0] == '-':
6         start_idx = 1
7         is_negative = True
8     else:
9         start_idx = 0
10        is_negative = False
11
12    for i in range(start_idx, len(input_str)):
13        place = 10**(len(input_str) - (i+1))
14        digit = ord(input_str[i]) - ord('0')
15        output_int += place * digit
16
17    if is_negative:
18        return -1 * output_int
19    else:
20        return output_int
21
22
```



```
23
24 s = "554"
25 x = str_to_int(s)
26 print(type(x))
27
28 s = "123"
29 print(str_to_int(s))
30
31 s = "-123"
32 print(str_to_int(s))
```



Output

1.74s

```
<class 'int'>
123
-123
```

Explanation

On **line 3**, `output_int` is initialized to `0`. The code on **lines 5-10** deal with the polarity of the number represented by `input_str`:

```
if input_str[0] == '-':
    start_idx = 1
    is_negative = True
else:
    start_idx = 0
    is_negative = False
```

If the first character of `input_str` is `-`, `start_idx` is set to `1` and `is_negative` is set to `True` (**lines 6-7**). `start_idx` is to indicate that we'll start processing `input_str` from the first index as `-` is on the zeroth index.

However, if `input_str[0]` is not equal to `-`, `start_idx` and `is_negative` are set to `0` and `False` respectively on **lines 9-10**.



Next, let's discuss the `for` loop on **line 12**:

```
for i in range(start_idx, len(input_str)):
    place = 10**(len(input_str) - (i+1))
    digit = ord(input_str[i]) - ord('0')
    output_int += place * digit
```

This `for` loop will run from `start_idx` to `len(input_str) - 1`. On **line 13**, `place` is set to `10` raised to the power of `len(input_str) - (i+1)`. For example, if `input_str` is `123`, `place` will have the following values:

```
input_str = 123
len(input_str) = 3

i = 0
place = 10** (3 - (0+1)) = 10** (2) = 100
i = 1
place = 10** (3 - (1+1)) = 10** (1) = 10
i = 2
place = 10** (3 - (2+1)) = 10** (0) = 1
```

You may notice that `place` is the base power calculated according to the place of the digit in the number. For the unit place, `place` will equal `1`. Once we have calculated the base power, we need to extract the digit as a number. Therefore, **on line 13**, we get the digit as an integer type by subtracting the Unicode code point of `0` from the Unicode code point of `input_str[i]`. The Unicode code points are calculated using the `ord` function.

Finally, all that is left for us is to multiply `digit` with `place` and add it to `output_int` which is done on **line 15**.

Lines 17-20 return `output_int` with its corresponding polarity.

```
if is_negative:
    return -1 * output_int
else:
    return output_int
```



If `is_negative` is `True`, `output_int` is returned after multiplication with `-1` on **line 18**; otherwise, it is returned as it is on **line 20**.

That's all, folks! Now we have come to an end to the course. I hope you had an amazing learning experience!

Did you complete this lesson?

YES!



← Back

Exercise: String to Integer



Report an
Issue



Ask a Question

(https://discuss.educative.io/tag/solution-review-string-to-integer__string-processing__data-structures-and-algorithms-in-python)

