



You're working on a secret team solving coded transmissions.

Your team is scrambling to decipher a recent message, worried it's a plot to break into a major European National Cake Vault. The message has been *mostly* deciphered, but all the words are backward! Your colleagues have handed off the last step to you.

Write a function `reverse_words()` that takes a message as a list of characters and reverses the order of the words in place.¹

Why a list of characters instead of a string?

The goal of this question is to practice manipulating strings *in place*. Since we're modifying the message, we need a mutable¹ type like a list, instead of Python 3.6's *immutable* strings.

For example:

```
message = [ 'c', 'a', 'k', 'e', ' ',  
            'p', 'o', 'u', 'n', 'd', ' ',  
            's', 't', 'e', 'a', 'l' ]
```

Python 3.6 ▼

```
reverse_words(message)
```

```
# Prints: 'steal pound cake'
```

```
print(''.join(message))
```

When writing your function, assume the message contains only letters and spaces, and all words are separated by one space.

Gotchas

We can do this in $O(1)$ space. Remember, in place¹.

We can do this in $O(n)$ time.

If you're swapping individual words one at a time, consider what happens when the words are different lengths. Isn't *each swap* $O(n)$ time in the worst case?

Breakdown

Let's start with a simpler problem. What if we wanted to **reverse all the characters** in the message?

Well, we could swap the first character with the last character, then the second character with the second to last character, and so on, moving towards the middle. Can you implement this in code?

```
def reverse_characters(message):  
    left_index = 0  
    right_index = len(message) - 1  
  
    # Walk towards the middle, from both sides  
    while left_index < right_index:  
        # Swap the left char and right char  
        message[left_index], message[right_index] = \  
            message[right_index], message[left_index]  
        left_index += 1  
        right_index -= 1
```

Python 3.6 ▼

We're using a cute one-liner to do the swap. In other languages you might need to do the swap by hand, recording one of the values in a temp variable.

Ok, looks good. **Does this help us?**

Can we use the same concept but apply it to *words* instead of *characters*?

Probably. We'll have to figure out a couple things:

1. How do we figure out where words begin and end?
2. Once we know the start and end indices of two words, how do we *swap* those two words?

We could attack either of those first, but I'm already seeing a potential problem in terms of runtime. Can you guess what it is?

What happens when you swap two words that *aren't* the same length?

```
# the eagle has landed
[ 't', 'h', 'e', ' ', 'e', 'a', 'g', 'l', 'e', ' ',
  'h', 'a', 's', ' ', 'l', 'a', 'n', 'd', 'e', 'd' ]
```

Python 3.6 ▼

Supposing we already knew the start and end indices of 'the' and 'landed', how long would it take to swap them?

$O(n)$ time, where n is the total length of the message!

Why? Notice that in addition to moving 'the' to the back and moving 'landed' to the front, we have to "scoot over" *everything in between*, since 'landed' is longer than 'the'.

So what'll be the *total* time cost with this approach? Assume we'll be able to learn the start and end indices of all of our words in just one pass ($O(n)$ time).

$O(n^2)$ total time. Why? In the worst case we have almost as many words as we have characters, and we're always swapping words of different lengths. For example:

```
# a bb c dd e ff g hh
[ 'a', ' ', 'b', 'b', ' ', 'c', ' ', 'd', 'd', ' ',
  'e', ' ', 'f', 'f', ' ', 'g', ' ', 'h', 'h' ]
```

Python 3.6 ▼

We take $O(n)$ time to swap the first and last words (we have to move all n characters):

```
# Input: a bb c dd e ff g hh
[ 'a', ' ', 'b', 'b', ' ', 'c', ' ', 'd', 'd', ' ',
  'e', ' ', 'f', 'f', ' ', 'g', ' ', 'h', 'h' ]

# First swap: hh bb c dd e ff g a
[ 'h', 'h', ' ', 'b', 'b', ' ', 'c', ' ', 'd', 'd',
  ' ', 'e', ' ', 'f', 'f', ' ', 'g', ' ', 'a' ]
```

Python 3.6 ▼

Then for the second swap:

```
# Input: a bb c dd e ff g hh
[ 'a', ' ', 'b', 'b', ' ', 'c', ' ', 'd', 'd', ' ', 'e', ' ', 'f', 'f', ' ', 'g', ' ', 'h', 'h' ]

# First swap: hh bb c dd e ff g a
[ 'h', 'h', ' ', 'b', 'b', ' ', 'c', ' ', 'd', 'd', ' ', 'e', ' ', 'f', 'f', ' ', 'g', ' ', 'a' ]

# Second swap: hh g c dd e ff bb a
[ 'h', 'h', ' ', 'g', ' ', 'c', ' ', 'd', 'd', ' ', 'e', ' ', 'f', 'f', ' ', 'b', 'b', ' ', 'a' ]
```

We have to move all n characters *except* the first and last words, and a couple spaces. So we move $n - 5$ characters in total.

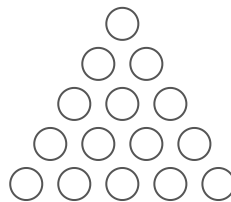
For the third swap, we have another 5 characters we don't have to move. So we move $n - 10$ in total. We'll end up with a series like this:

$$n + (n - 5) + (n - 10) + (n - 15) + \dots + 6 + 1$$

This is a subsection of the common triangular series.¹

A **triangular series** is a series of numbers where each number could be the row of an equilateral triangle.

So 1, 2, 3, 4, 5 is a triangular series, because you could stack the numbers like this:



Their sum is 15, which makes 15 a **triangular number**.

A triangular series *always* starts with 1 and increases by 1 with each number.

Since the only thing that changes in triangular series is the value of the highest number, it's helpful to give that a name. Let's call it n .

```
# n is 8
```

```
1, 2, 3, 4, 5, 6, 7, 8
```

Triangular series are nice because no matter how large n is, it's always easy to find the total sum of all the numbers.

Take the example above. Notice that if we add the first and last numbers together, and then add the second and second-to-last numbers together, they have the same sum! This happens with *every* pair of numbers until we reach the middle. If we add up all the pairs of numbers, we get:

$$1 + 8 = 9$$

$$2 + 7 = 9$$

$$3 + 6 = 9$$

$$4 + 5 = 9$$

This is true for *every triangular series*:

1. **Pairs of numbers** on each side will always **add up to the same value**.
2. That value will always be **1 more than the series' n** .

This gives us a pattern. Each pair's sum is $n + 1$, and there are $\frac{n}{2}$ pairs. So our total sum is:

$$(n + 1) * \frac{n}{2}$$

Or:

$$\frac{n^2 + n}{2}$$

Ok, but does this work with triangular series with an *odd* number of elements? Yes. Let's say $n = 5$. So if we calculate the sum by hand:

$$1 + 2 + 3 + 4 + 5 = 15$$

And if we use the formula, we get the same answer:

$$\frac{5^2 + 5}{2} = 15$$

One more thing:

What if we know *the total sum*, but we *don't* know the value of *n*?

Let's say we have:

$$1 + 2 + 3 + \dots + (n - 2) + (n - 1) + n = 78$$

No problem. We just use our formula and set it equal to the sum!

$$\frac{n^2+n}{2} = 78$$

Now, we can rearrange our equation to get a *quadratic equation* (remember those?)

$$n^2 + n = 156$$

$$n^2 + n - 156 = 0$$

Here's the quadratic formula:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

If you don't really remember how to use it, that's cool. You can just use an online calculator. We don't judge.

Taking the positive solution, we get $n = 12$.

So for a triangular series, remember—the total sum is:

$$\frac{n^2+n}{2}$$

We're just skipping 4 terms between each term!

So we have the sum of "every fifth number" from that triangular series. That means our sum will be about a fifth the sum of the original series! But in big O notation dividing by 5 is a constant, so we can throw it out. The original triangular series is $O(n^2)$, and so is our series with every fifth element!

Okay, so $O(n^2)$ time. That's pretty bad. It's *possible* that's the best we can do...but maybe we can do better?

Let's try manipulating a sample input by hand.

And remember what we did for our character-level reversal...

Look what happens when we do a character-level reversal:

```
# Input: the eagle has landed
[ 't', 'h', 'e', ' ', 'e', 'a', 'g', 'l', 'e', ' ',
  'h', 'a', 's', ' ', 'l', 'a', 'n', 'd', 'e', 'd' ]

# Character-reversed: dednal sah elgae eht
[ 'd', 'e', 'd', 'n', 'a', 'l', ' ', 's', 'a', 'h', ' ',
  'e', 'l', 'g', 'a', 'e', ' ', 'e', 'h', 't' ]
```

Python 3.6 ▼

Notice anything?

What if we put it up against the desired output:

```
# Input: the eagle has landed
[ 't', 'h', 'e', ' ', 'e', 'a', 'g', 'l', 'e', ' ',
  'h', 'a', 's', ' ', 'l', 'a', 'n', 'd', 'e', 'd' ]

# Character-reversed: dednal sah elgae eht
[ 'd', 'e', 'd', 'n', 'a', 'l', ' ', 's', 'a', 'h', ' ',
  'e', 'l', 'g', 'a', 'e', ' ', 'e', 'h', 't' ]

# Word-reversed (desired output): landed has eagle the
[ 'l', 'a', 'n', 'd', 'e', 'd', ' ', 'h', 'a', 's', ' ',
  'e', 'a', 'g', 'l', 'e', ' ', 't', 'h', 'e' ]
```

Python 3.6 ▼

The character reversal reverses the words! It's a great first step. From there, we just have to "unscramble" each word.

More precisely, we just have to re-reverse each word!

Solution

We'll write a helper function `reverse_characters()` that reverses all the characters between a `left_index` and `right_index`. We use it to:

1. Reverse **all the characters in the entire message**, giving us the correct *word order* but with *each word backward*.
2. Reverse **the characters in each individual word**.

Python 3.6 ▼

```
def reverse_words(message):  
    # First we reverse all the characters in the entire message  
    reverse_characters(message, 0, len(message)-1)  
  
    # This gives us the right word order  
    # but with each word backward  
  
    # Now we'll make the words forward again  
    # by reversing each word's characters  
  
    # We hold the index of the *start* of the current word  
    # as we look for the *end* of the current word  
    current_word_start_index = 0  
  
    for i in range(len(message) + 1):  
        # Found the end of the current word!  
        if (i == len(message)) or (message[i] == ' '):  
            reverse_characters(message, current_word_start_index, i - 1)  
            # If we haven't exhausted the message our  
            # next word's start is one character ahead  
            current_word_start_index = i + 1  
  
def reverse_characters(message, left_index, right_index):  
    # Walk towards the middle, from both sides  
    while left_index < right_index:  
        # Swap the left char and right char  
        message[left_index], message[right_index] = \  
            message[right_index], message[left_index]  
        left_index += 1  
        right_index -= 1
```

Complexity

$O(n)$ time and $O(1)$ space!

Hmm, the team used your function to finish deciphering the message. There definitely seems to be a heist brewing, but no specifics on where. Any ideas?

Bonus

How would you handle punctuation?

What We Learned

The naive solution of reversing the words one at a time had a worst-case $O(n^2)$ runtime. That's because swapping words with *different lengths* required "scooting over" all the other characters to make room.

To get around this "scooting over" issue, we reversed all the *characters* in the message first. This put all the words in the correct spots, but with the characters in each word backward. So to get the final answer, we reversed the characters in each word. This all takes two passes through the message, so $O(n)$ time total.

This might seem like a blind insight, but we derived it by using a clear strategy:

Solve a *simpler* version of the problem (in this case, reversing the characters instead of the words), and see if that gets us closer to a solution for the original problem.

We talk about this strategy in the "get unstuck" section of our coding interview tips (</article/coding-interview-tips#unstuck>).

Ready for more?

Check out our full course ➡

Want more coding interview help?

Check out **interviewcake.com** for more advice, guides, and practice questions.