49

~ ~
 ~
~ ~
 ~
~ ~
 ~
~ ~
 ~
~ ~
 ~
~ ~
 ~
~ ~
 ~
~ ~
 ~
~ ~
 ~
~ ~
 ~
~ ~
 ~
~ ~
 ~
~ ~
 ~
~ ~
 ~
~ ~
 ~
~ ~
 ~
~ ~
 ~
~ ~
 ~
~ ~
 ~
~ ~
 ~
~ ~
 ~
~ ~ an ongoing artistic journal in digitally published zines ~
 ~
~ ~
 ~
~ ~
 ~
~ ~
 ~
~ ~
 ~
~ ~

esther seyffarth
how not to reverse a string

esther seyffarth
how not to reverse a string

Intro

=======

Some questions in life have a single, canonical, unequivocally correct answer: What's one plus one? Who's a good boy? How do you reverse a string in this programming language? For the last question, the pythonic answer is this:

```
>>> print('):'[::-1])
:)
```

But dig a little deeper, and you'll soon find that an ostensibly simple question can have an infinite spectrum of solutions, ranging all the way from obvious to deranged. Now that you know what the straightforward solution is, you're free to look for all the non-straightforward, non-simple, non-traditional approaches to the problem.

This is a collection of roundabout, inefficient, overly-complex, needlessly elaborate approaches to the task of reversing strings in Python. Be advised that the code examples you will find here are not very good, and not at all recommended for use at home!

#01

```
def format_reverse(in_word):
   out_word = '{}' * len(in_word)
   for i in range(len(in_word)):
       rest = ['{}' for j in range(len(in_word) - i-1)]
       out_word = out_word.format(*rest, in_word[i])
       print(out_word)
   return out_word
```

```
>>> format_reverse('bad idea')
{}{}{}{}{}{}{}b
{}{}{}{}{}{}ab
{}{}{}{}{}dab
{}{}{}{} dab
{}{}{}i dab
{}{}di dab
{}edi dab
aedi dab
aedi dab
```

This is a bad idea. We're creating a sequence of placeholders, and then filling them with characters from the input string, starting with the first char and the last placeholder, and going all the way to the last char and first placeholder. Don't do this at home, unless you really, really like curly braces.

~ ~

#02

```
import numpy as np
def numpy_reverse(word):
    out = np.empty([len(word), len(word)], dtype=np.unicode_)
    out[0,] = np.asarray(list(word), dtype=np.unicode_)
    print(out)
    out = np.rot90(out)
    print(out)
    out = out.transpose(1,0)
    return ''.join([c for c in out[0,]])
```

```
>>> numpy_reverse('joy')
[['j' 'o' 'y']
 ['' '' '']
 ['' '' '']]
[['y' '' '']
 ['o' '' '']
 ['j' '' '']]
[['y' 'o' 'j']
 ['' '' '']
 ['' '' '']]
yoj
```

Imagine treating every problem you encounter as a math problem. Every single thing. Now imagine you have at your disposal a really neat tool for solving mathematical problems. The logical next step is this code. It creates an array that contains the input string, then rotates that array by 90 degrees, and finally transposes the result to give you the reversed string. Math!

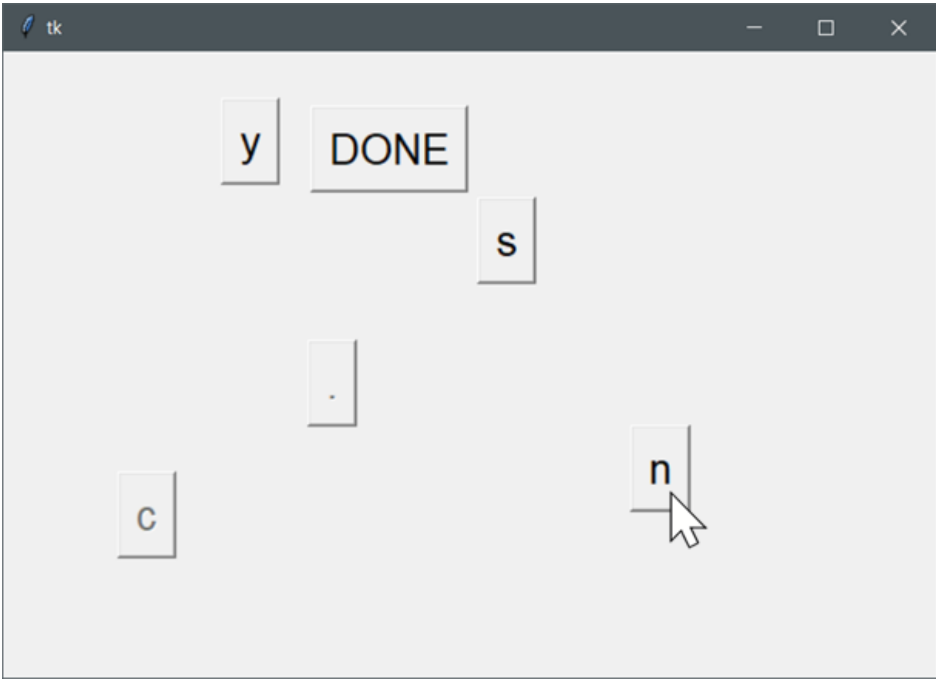
```
def build_tree(word):
    if len(word) == 1:
        return word
    return [word[0], build_tree(word[1:])]

def tree_swap (tree):
    return list(tree[-1]) + list(tree[0])

def tree_reverse(word):
    tree = build_tree(word)
    print('Initial tree: {}\n'.format(tree))
    return ''.join(tree_walk(tree))

def tree_walk(tree):
    print(tree)
    if len(tree[1]) == 1:
        tree = tree_swap(tree)
    else:
        tree = tree_walk(tree[-1]) + list(tree[0])
    print(tree)

    return tree
```

```
>>> tree_reverse('flippy tree')

Initial tree: ['f', ['l', ['i', ['p', ['p', ['y', [' ', ['t', ['r',
['e', 'e']]]]]]]]]]

['f', ['l', ['i', ['p', ['p', ['y', [' ', ['t', ['r', ['e',
'e']]]]]]]]]]
['l', ['i', ['p', ['p', ['y', [' ', ['t', ['r', ['e', 'e']]]]]]]]]
['i', ['p', ['p', ['y', [' ', ['t', ['r', ['e', 'e']]]]]]]]
['p', ['p', ['y', [' ', ['t', ['r', ['e', 'e']]]]]]]
['p', ['y', [' ', ['t', ['r', ['e', 'e']]]]]]
['y', [' ', ['t', ['r', ['e', 'e']]]]]
[' ', ['t', ['r', ['e', 'e']]]]
['t', ['r', ['e', 'e']]]
['r', ['e', 'e']]
['e', 'e']
['e', 'e']
['e', 'e', 'r']
['e', 'e', 'r', 't']
['e', 'e', 'r', 't', ' ']
['e', 'e', 'r', 't', ' ', 'y']
['e', 'e', 'r', 't', ' ', 'y', 'p']
['e', 'e', 'r', 't', ' ', 'y', 'p', 'p']
['e', 'e', 'r', 't', ' ', 'y', 'p', 'p', 'i']
['e', 'e', 'r', 't', ' ', 'y', 'p', 'p', 'i', 'l']
['e', 'e', 'r', 't', ' ', 'y', 'p', 'p', 'i', 'l', 'f']
eert yppilf
```

Treat a word like a tree. Each character branches off from the rest of the word. Now take that tree, and spin it around: Each branching point is still a branching point, but now the direction is changed. Beautiful, but complicated.

```
import tkinter as tk
import random
REV = ''

def gui_reverse(word):
    print('Please select the characters from the string in ' +
          'their correct order.')
    root = tk.Tk()
    canvas = tk.Canvas(root, width=600, height=400)
    canvas.pack(fill='both', expand=True)
    for c in word:
        label = tk.Button(root, text=c, font=('Helvetica', '20'))
        place_label(canvas, label)
    tk_exit = tk.Button(root, text='DONE', font=('Helvetica', '20'),
                        command=lambda:close(root))
    place_label(canvas, tk_exit)
    root.mainloop()

def place_label(canvas, label):
    if label['text'] != 'DONE':
        label['command'] = lambda: add_clicked_char(label)

    # This part taken from stackoverflow.com/a/52526584:
    # Place a label on a canvas in a random, non-overlapping location
    width = label.winfo_reqwidth()
    height = label.winfo_reqheight()
    tries = 0
    while True and tries < 1000:
        tries += 1
        x = random.randint(0, 600-width)
        y = random.randint(0, 400-height)
        items = canvas.find_overlapping(x, y, x+width, y+height)
        if len(items) == 0:
            canvas.create_window(x, y, window=label, anchor='nw')
            break

def add_clicked_char(label):
    global REV
    REV += label['text']
    label['state'] = 'disabled'

def close(window):
    global REV
    print(("'\nYou decided to reverse this string:\n'{}'\n" +
           "You selected this sequence to do so:\n'{}'\n\n" +
           "Thank you for your time.").format(word, REV))
    window.destroy()

# see next page for output............ ==>
```

```
>>> gui_reverse('sync.')

Please select the characters from the string in their correct order.
```




```
You decided to reverse this string:
'sync.'
You selected this sequence to do so:
'.cnys'

Thank you for your time.
```

Computers make things easier, but it doesn't have to be that way. We can also use computers to make a task several times more difficult than it has to be, like we did here. This is not helping anyone. This approach to the task is unnecessarily mean, the user has every right to do a table flip or at least step away from the screen and spend their time doing literally anything else instead. Of all the solutions to reversing strings presented in this collection, this is definitely the least recommended one. Never ever do this, except maybe when you're planning on pranking someone.

#05

```
class Stack_Reverser():
    def __init__(self, input_word):
        self.original_word = input_word
        self.stack = []

    def is_empty(self):
        return not bool(len(self.stack))

    def add_element(self, el):
        self.stack.append(el)

    def remove_element(self):
        return self.stack.pop()

    def reverse_string(self):
        out = ""

        for c in self.original_word:
            self.add_element(c)
        print(self.stack)

        while not self.is_empty():
            out += self.remove_element()
            print(self.stack)
            print(out)

        return out
```

```
>>> s = Stack_Reverser('not good')
>>> s.reverse_string()

['n', 'o', 't', ' ', 'g', 'o', 'o', 'd']
['n', 'o', 't', ' ', 'g', 'o', 'o']
d
['n', 'o', 't', ' ', 'g', 'o']
do
['n', 'o', 't', ' ', 'g']
doo
['n', 'o', 't', ' ']
doog
['n', 'o', 't']
doog
['n', 'o']
doog t
['n']
doog to
[]
doog ton
doog ton
```

Listen. I apologize. Some of the code examples I've given so far are slightly off-putting, and I am sorry. Here's a real good way to reverse strings. Store your characters on a stack, and then grab them from the stack one by one, starting from the top, to stick together the solution. Simple! Stylish! Fun!

#06

```
def wrap_reverse(word):
    word = list(word)
    for i in range(len(word)):
        print(''.join(word))
        new_i = len(word) - i-1
        for j in range(0, new_i):
            word[j], word[j+1] = word[j+1], word[j]
    return ''.join(word)
```

```
>>> wrap_reverse('wrap this')

wrap this
rap thisw
ap thisrw
p thisarw
 thisparw
this parw
hist parw
isht parw
siht parw
siht parw
```

Here are two solutions in which each letter travels through the input word until it reaches its new index. In this one, we always move the first char, until we've moved all the characters that are contained in the input string...

~ ~

#07

Or how about moving two characters at the same time, like here? That way, we get to stop as soon as we get to the middle of the word! This drastically reduces the complexity of the algorithm.

```
>>> swap_reverse('swap that')

swap that
twap thas
taap thws
tahp taws
taht paws
taht paws
```

```
def swap_reverse(word):
    print(word)
    for i in range(len(word) // 2):
        start = word[:i] + word[-i-1]
        middle = word[i+1:-i-1]
        end = word[i] + int(bool(i)) * word[-i:]
        word = start + middle + end
        print(word)
    return word
```

```
def bit_reverse(input_word):
    output = 0
    for i in range(len(input_word)):
        char = input_word[i]
        shift = 8 * i
        output = output | (int(bin(ord(char)), 2) << shift)

        print('{:<30}{}'.format('Current char:', char))
        print('{:<30}{}'.format('Current char (binary):',
                                bin(ord(char))))
        print('{:<30}{}'.format('Current output (binary):',
                              bin(output)))
        print('---')

    print('\n##### AND NOW, BACKWARDS #####\n')

    loop_start_index = len(bin(ord(char)))
    out_word = char
    for i in range(loop_start_index, len(bin(output)), 8):
        out_word += chr(int(bin(output)[i:i+8], 2))

        print('{:<35}{}'.format('Current char (binary):',
                                bin(output)[i:i+8]))
        print('{:<35}{}'.format('Current char (integer base 10):',
                                int(bin(output)[i:i+8], 2)))
        print("{:<35}'{}'".format('Current char as character:',
                        chr(int(bin(output)[i:i+8], 2))))
        print('---')

    return out_word

# see next page for output............ ==>
```

Here's how to reverse a string if you're secretly much more excited about numbers than you are about words. The process is straight-forward and contains these steps:

1. *For each character, determine its utf8 code.*
2. *Collect all utf8 representations by bit-shifting every new character to the left and connecting it to the results so far with a bitwise OR.*
3. *At this point, you have a huge sequence of ones and zeroes that contains your input string. That's cool!*
4. *We can now read that sequence from the left and chunk it up into its parts. The leftmost binary chunk we get is the last character of the string, the rightmost chunk that's left will be the first character.*
5. *Let's turn the binary code back to characters for slightly better human readability.*
6. *You've successfully reversed a string using binary magic.*

```
>>> bit_reverse('¿wtf?')

Current char:                 ¿
Current char (binary):        0b10111111
Current output (binary):      0b10111111
---
Current char:                 w
Current char (binary):        0b1110111
Current output (binary):      0b111011110111111
---
Current char:                 t
Current char (binary):        0b1110100
Current output (binary):      0b1110100111011110111111
---
Current char:                 f
Current char (binary):        0b1100110
Current output (binary):      0b11001101110100111011110111111
---
Current char:                 ?
Current char (binary):        0b111111
Current output (binary):
0b11111101100110111010011101110111111
---

##### AND NOW, BACKWARDS #####

Current char (binary):             01100110
Current char (integer, base 10):   102
Current char as character:         'f'
---
Current char (binary):             01110100
Current char (integer, base 10):   116
Current char as character:         't'
---
Current char (binary):             01110111
Current char (integer, base 10):   119
Current char as character:         'w'
---
Current char (binary):             10111111
Current char (integer, base 10):   191
Current char as character:         '¿'
---
?ftw¿
```

```
def learn_reverse():

    known_reverses = {}
    w = input('What would you like to reverse?\n> ')

    while w:
        if w in known_reverses:
            print("The reverse of '{}' is '{}'.".format(w,
                                           known_reverses[w]))
        else:
            t = input('Sorry, I don't know that string. ' +
                      'Please tell me how to reverse it.\n> ')
            if not t:
                break

            known_reverses[w] = t
        w = input('What would you like to reverse?\n> ')

    print('Goodbye! Thank you for teaching me! ')
```

```
>>> learn_reverse()

What would you like to reverse?
> sync
Sorry, I don't know that string. Please
tell me how to reverse it.
> cnys
What would you like to reverse now?
> zine
Sorry, I don't know that string. Please
tell me how to reverse it.
> eniz
What would you like to reverse now?
> sync
The reverse of 'sync' is 'cnys'.
What would you like to reverse now?
>
Goodbye! Thank you for teaching me!
```

Maybe the real reversed string was inside you all along. This solution is an attempt to figure out whether that's true. After all, a computer program is only ever as smart as the person who wrote it, so it makes sense that this program is extra cautious and tries to only give answers it knows you'll like.

Not recommended if you're in a hurry. Also not recommended in any other scenario.

#10

```
def recursive_reverse(word):

    if len(word) == 1 or len(word) == 0:
        print(word)
        return word

    else:
        print(word)
        return '{}{}{}'.format(word[-1],
                               recursive_reverse(word[1:-1]),
                               word[0])
```

```
>>> recursive_reverse('recursion!')

recursion!
ecursion
cursio
ursi
rs

!noisrucer
```

Recursion. A simple, yet inefficient solution to string-reversing. Guaranteed to win you points in a programming class, as well as barely anywhere else. This is the slowest of the solutions presented here (if you factor out all the ones that depend on human intervention).

~ ~

#11

```
def rec_rev(w): return (len(w)==1 and w) or rec_rev(w[1:])+w[0]
```

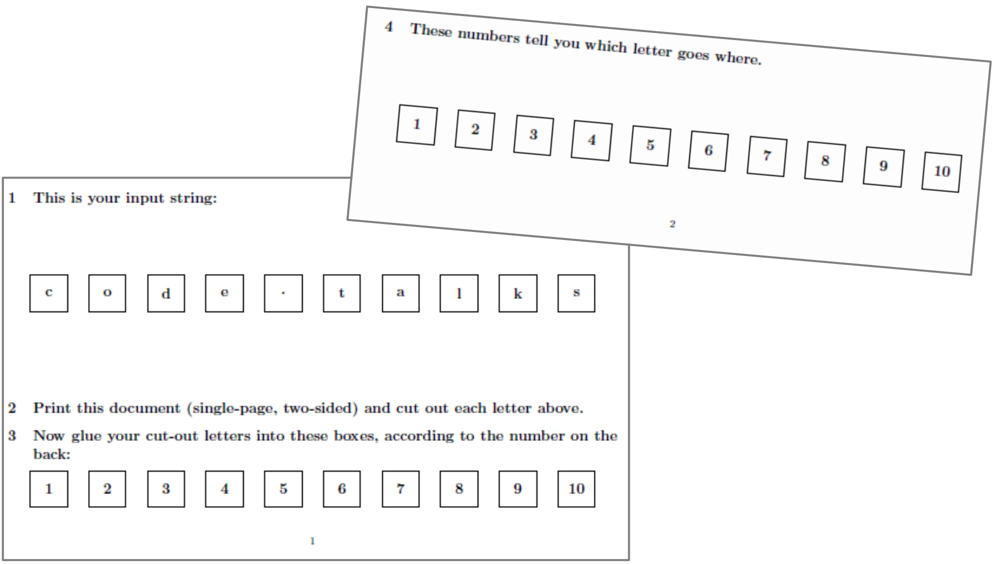
If you're short on space, you could do worse than this neat one-liner solution. The logic is basically the same as above, with a twist:

```
>>> rec_rev('):')

:)
```

The evaluation of Boolean operations in Python results in different values, depending on where the partial expression that evaluates to "False" is located. The examples to the right illustrate this behavior. Thanks, Python!

```
>>> 'spaghetti' or True
spaghetti
>>> True or 'spaghetti'
True
>>> 'spaghetti' and True
True
>>> True and 'spaghetti'
spaghetti
```

```
def tex_reverse(word):
    # ...

    # Let's generate a LaTeX file! See next page for more... ==>

    node_1 = '\\node[rectangle, draw=black, very thick, ' +
             'inner sep=8pt, minimum size=3\\baselineskip] ' +
             ' ({}) at (0, 0) {{\\Large\\textbf{{{}}}}};'
    node_2 = '\\node[rectangle, draw=black, very thick, ' +
             'inner sep=8pt, minimum size=3\\baselineskip] ' +
             '({}) at ({}, 0) {{\\Large\\textbf{{{}}}}};'

    # sn1, sn2, sn3, sn4 are LaTeX snippets defined separately
    with open(outpath, 'w', encoding='utf8') as out:
        print(sn1, file=out)
        print(node_1.format(word[0], word[0]), file=out)

        for i in range(1, len(word)):
            print(node_2.format(word[i], i*2, word[i]), file=out)
        print(sn2, file=out)
        print(node_1.format(1, 1), file=out)

        for i in range(1, len(word)):
            print(node_2.format(i+1, i*2, i+1), file=out)
            print(sn3, file=out)
            print(node_1.format(1, 1), file=out)
            for i in range(1, len(word)):
                print(node_2.format(i+1, i*2, i+1), file=out)

      print(sn4, file=out)

# Compile the resulting .tex file to create the pdf shown below!
```



4 These numbers tell you which letter goes where.

1	2	3	4	5	6	7	8	9	10

2

1 This is your input string:

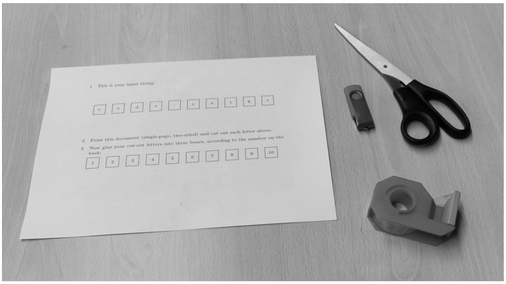
c	o	d	e	.	t	a	l	k	s

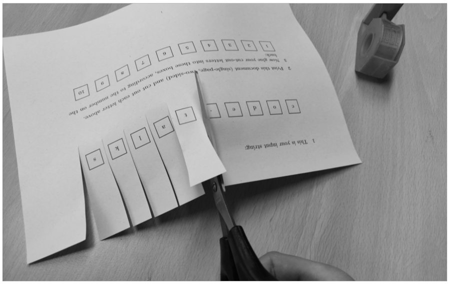
2 Print this document (single-page, two-sided) and cut out each letter above.

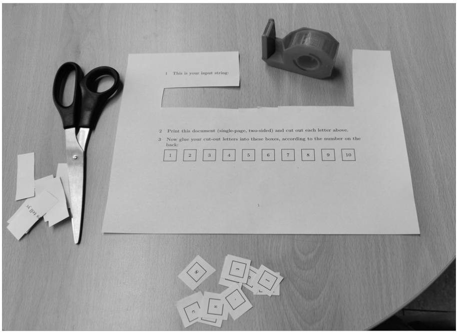
3 Now glue your cut-out letters into these boxes, according to the number on the back:

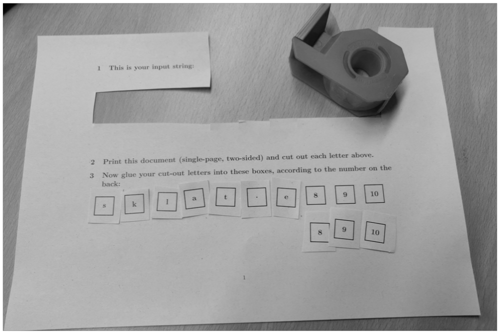
1	2	3	4	5	6	7	8	9	10

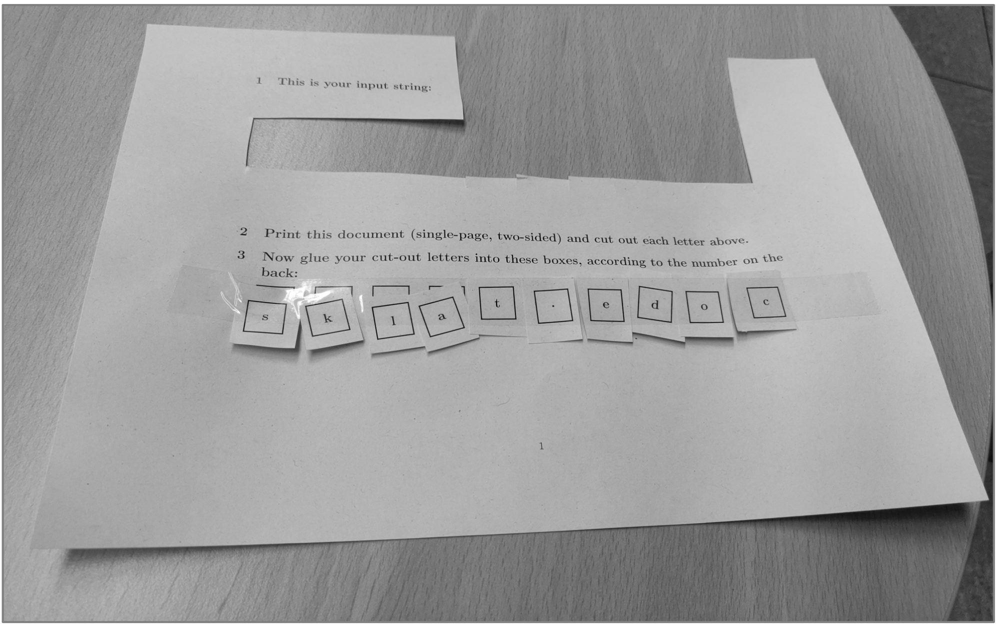
1


1 This is your input string:
2 Print this document (single-page, two-sided) and cut out each letter above.
3 Now glue your cut-out letters into these boxes, according to the number on the back:
1 2 3 4 5 6 7 8 9 10




1 This is your input string:
2 Print this document (single-page, two-sided) and cut out each letter above.
3 Now glue your cut-out letters into these boxes, according to the number on the back:
1 2 3 4 5 6 7 8 9 10


1 This is your input string:
2 Print this document (single-page, two-sided) and cut out each letter above.
3 Now glue your cut-out letters into these boxes, according to the number on the back:
s k l a t . e 8 9 10
8 9 10
1


1 This is your input string:
2 Print this document (single-page, two-sided) and cut out each letter above.
3 Now glue your cut-out letters into these boxes, according to the number on the back:
s k l a t . e d o c
1

```
import random

def ask_reverse(word):
    out = ''

    while len(out) < len(word):
        next = random.randint(0, len(word)-1)
        is_this_it = input(('You entered this word: \t{}\n' +
                            'Our output so far is this:\t{}\n' +
                            'Add this char (y/n)?\t{}\n>>> ').
                            format(word, out, word[next]))
        if is_this_it.lower() in ['yes', 'y']:
            out += word[next]

    return 'Here is your result:\t{}'.format(out)
```

```
>>> ask_reverse('Matt')

You entered this word:  Matt
Our output so far is this:
Should I add this char (y/n)?   t
>>> y
You entered this word:  Matt
Our output so far is this:      t
Should I add this char (y/n)?   t
>>> y
You entered this word:  Matt
Our output so far is this:      tt
Should I add this char (y/n)?   a
>>> y
You entered this word:  Matt
Our output so far is this:      tta
Should I add this char (y/n)?   a
>>> n
You entered this word:  Matt
Our output so far is this:      tta
Should I add this char (y/n)?   a
>>> n
You entered this word:  Matt
Our output so far is this:      tta
Should I add this char (y/n)?   a
>>> n
You entered this word:  Matt
Our output so far is this:      tta
Should I add this char (y/n)?   t
>>> n
You entered this word:  Matt
Our output so far is this:      tta
Should I add this char (y/n)?   M
>>> y
Here is your result:    ttaM
```

This solution was proposed by Matt Schneider (Twitter: @Mattlaschneider).

The program is doing its best, but its best is not very good, so it decides to ask you what it should add to the output word in each step. It does not remember which chars it has already placed. It really needs you, so please be kind to it.

This approach becomes worse as the length of the input word grows, so maybe don't attempt to reverse Moby Dick with it (but also, don't choose any of the other solutions in this collection for that task).

```
import random

def slow_rev(word):
    out = ''
    while word:
        i = random.randint(0, len(word)-1)
        print(word)
        if i == len(word) - 1:
            out += word[i]
            word = word[:i]
            print('output: {}'.format(out))

    return out
```

```
>>> slow_rev('bad')

bad
bad
new word: d
ba
ba
ba
ba
new word: da
b
new word: dab
dab
```

Add a random character to the output, but only after checking that it's a good idea to add that character. Don't let the simple structure of this code fool you: The algorithm is horrible, even though it looks kind of nice from the outside.

~ ~

#15

```
# idea: twitter.com/miseryconfusion
import time
import sys
from threading import Thread

def sleep_append(delay, el, output):
  delay = delay/5
  time.sleep(delay)
  print(output)
  output.append(el)

def sleep_reverse(l):
  output = []
  threads = [Thread(target=sleep_append,
             args=(len(l)-i, v, output)) \
             for i, v in enumerate(l)]
  for t in threads:
    t.start()
  for t in threads:
    t.join()

  return ''.join(output)
```

```
>>> sleep_reverse('slow')

[]
['w']
['w', 'o']
['w', 'o', 'l']
wols
```

This approach is based on waiting (even more than the others you've seen so far). We wait for each character, one after the other, and the shortest wait is assigned to the last character. The first character is the one we have to wait the longest for. Original idea by @miseryconfusion on Twitter.

user.phil.hhu.de/~seyffarth

sync.abue.io/

-----

p2p

191220es_sync2_49_reverses.pdf