Fast punctuation removal with pandas

Asked 6 years, 7 months ago Modified 2 years, 4 months ago Viewed 27k times

87

This is a self-answered post. Below I outline a common problem in the NLP domain and propose a few performant methods to solve it.

Oftentimes the need arises to remove **punctuation** during text cleaning and pre-processing. Punctuation is defined as any character in `string.punctuation`:

```
>>> import string
string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

This is a common enough problem and has been asked before ad nauseam. The most idiomatic solution uses pandas `str.replace`. However, for situations which involve a *lot* of text, a more performant solution may need to be considered.

What are some good, performant alternatives to `str.replace` when dealing with hundreds of thousands of records?

python regex string pandas numpy

Share Follow

edited Dec 28, 2018 at 12:21

asked May 21, 2018 at 7:58

cs95

402k ● 104 ● 735 ● 788

4 Answers

Sorted by: Highest score (default)

100

+250

Setup

For the purpose of demonstration, let's consider this DataFrame.

```
df = pd.DataFrame({'text':['a..b?!??', '%hgh&12','abc123!!!', '$$$1234']})
df
        text
0   a..b?!??
1    %hgh&12
2  abc123!!!
3    $$$1234
```

Below, I list the alternatives, one by one, in increasing order of performance

`str.replace`

This option is included to establish the default method as a benchmark for comparing other, more performant solutions.

This uses pandas in-built `str.replace` function which performs regex-based replacement.

```
df['text'] = df['text'].str.replace(r'[^\w\s]+', '')
```

```
df
     text
0      ab
1   hgh12
2  abc123
3    1234
```

This is very easy to code, and is quite readable, but slow.

`regex.sub`

This involves using the `sub` function from the `re` library. Pre-compile a regex pattern for performance, and call `regex.sub` inside a list comprehension. Convert `df['text']` to a list beforehand if you can spare some memory, you'll get a nice little performance boost out of this.

```
import re
p = re.compile(r'[^\w\s]+')
df['text'] = [p.sub('', x) for x in df['text'].tolist()]
```

```
df
     text
0      ab
1   hgh12
2  abc123
3    1234
```

Note: If your data has NaN values, this (as well as the next method below) will not work as is. See the section on "**Other Considerations**".

`str.translate`

python's `str.translate` function is implemented in C, and is therefore *very fast*.

How this works is:

1. First, join all your strings together to form one *huge* string using a single (or more) character **separator** that *you* choose. You *must* use a character/substring that you can guarantee will not belong inside your data.
2. Perform `str.translate` on the large string, removing punctuation (the separator from step 1 excluded).
3. Split the string on the separator that was used to join in step 1. The resultant list *must* have the same length as your initial column.

Here, in this example, we consider the pipe separator `|`. If your data contains the pipe, then you must choose another separator.

```
import string

punct = '!"#$%&\'()*+,-./:;<=>?@[\\]^_`{}~'   # `|` is not present here
transtab = str.maketrans(dict.fromkeys(punct, ''))

df['text'] = '|'.join(df['text'].tolist()).translate(transtab).split('|')
```

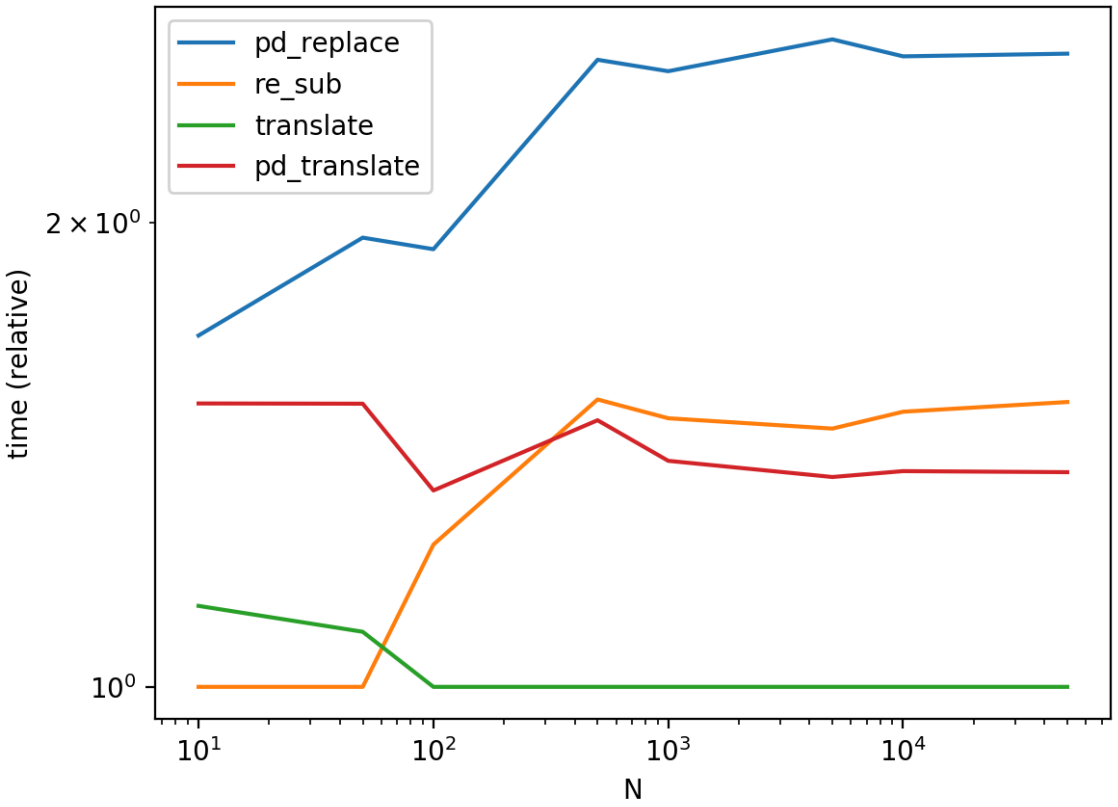
```
df
     text
0      ab
1   hgh12
2  abc123
3    1234
```

Performance

`str.translate` performs the best, by far. Note that the graph below includes another variant `Series.str.translate` from MaxU's answer.

(Interestingly, I reran this a second time, and the results are slightly different from before. During the second run, it seems `re.sub` was winning out over `str.translate`

for really small amounts of data.)



There is an inherent risk involved with using `translate` (particularly, the problem of *automating* the process of deciding which separator to use is non-trivial), but the trade-offs are worth the risk.

Other Considerations

Handling NaNs with list comprehension methods; Note that this method (and the next) will only work as long as your data does not have NaNs. When handling NaNs, you will have to determine the indices of non-null values and replace those only. Try something like this:

```
df = pd.DataFrame({'text': [
    'a..b?!??', np.nan, '%hgh&12','abc123!!!', '$$$1234', np.nan]})

idx = np.flatnonzero(df['text'].notna())
col_idx = df.columns.get_loc('text')
df.iloc[idx,col_idx] = [
    p.sub('', x) for x in df.iloc[idx,col_idx].tolist()]

df
     text
0      ab
1     NaN
2   hgh12
3  abc123
```

```
4    1234
5     NaN
```

Dealing with DataFrames; If you are dealing with DataFrames, where *every* column requires replacement, the procedure is simple:

```
v = pd.Series(df.values.ravel())
df[:] = translate(v).values.reshape(df.shape)
```

Or,

```
v = df.stack()
v[:] = translate(v)
df = v.unstack()
```

Note that the `translate` function is defined below in with the benchmarking code.

Every solution has tradeoffs, so deciding what solution best fits your needs will depend on what you're willing to sacrifice. Two very common considerations are performance (which we've already seen), and memory usage. `str.translate` is a memory-hungry solution, so use with caution.

Another consideration is the complexity of your regex. Sometimes, you may want to remove anything that is not alphanumeric or whitespace. Othertimes, you will need to retain certain characters, such as hyphens, colons, and sentence terminators `[.!?]`. Specifying these explicitly add complexity to your regex, which may in turn impact the performance of these solutions. Make sure you test these solutions on your data before deciding what to use.

Lastly, unicode characters will be removed with this solution. You may want to tweak your regex (if using a regex-based solution), or just go with `str.translate` otherwise.

For even *more* performance (for larger N), take a look at this answer by Paul Panzer.

Appendix

Functions

```
def pd_replace(df):
    return df.assign(text=df['text'].str.replace(r'[^\w\s]+', ''))


def re_sub(df):
    p = re.compile(r'[^\w\s]+')
    return df.assign(text=[p.sub('', x) for x in df['text'].tolist()])

def translate(df):
```

```
    punct = string.punctuation.replace('|', '')
    transtab = str.maketrans(dict.fromkeys(punct, ''))

    return df.assign(
        text='|'.join(df['text'].tolist()).translate(transtab).split('|')
    )

# MaxU's version (https://stackoverflow.com/a/50444659/4909087)
def pd_translate(df):
    punct = string.punctuation.replace('|', '')
    transtab = str.maketrans(dict.fromkeys(punct, ''))

    return df.assign(text=df['text'].str.translate(transtab))
```

Performance Benchmarking Code

```
from timeit import timeit

import pandas as pd
import matplotlib.pyplot as plt

res = pd.DataFrame(
       index=['pd_replace', 're_sub', 'translate', 'pd_translate'],
       columns=[10, 50, 100, 500, 1000, 5000, 10000, 50000],
       dtype=float
)

for f in res.index:
    for c in res.columns:
        l = ['a..b?!??', '%hgh&12','abc123!!!', '$$$1234'] * c
        df = pd.DataFrame({'text' : l})
        stmt = '{}(df)'.format(f)
        setp = 'from __main__ import df, {}'.format(f)
        res.at[f, c] = timeit(stmt, setp, number=30)

ax = res.div(res.min()).T.plot(loglog=True)
ax.set_xlabel("N");
ax.set_ylabel("time (relative)");

plt.show()
```

Share Follow

edited Feb 10, 2019 at 3:49

answered May 21, 2018 at 7:58

cs95

402k ● 104 ● 735 ● 788

2 Great explanation, thanks! Is it possible to extend this analysis / method to 1. removing stopwords 2. stemming words 3. making all words lowercase ? – PyRsquared May 31, 2018 at 14:29

2 @killerT2333 I've written somewhat of a blog post for it here in this answer. I hope you find it useful. Any feedback/criticism welcomed. – cs95 May 31, 2018 at 14:41

2 @killerT2333 Small note: that post doesn't involve actually calling the lemmatizer/stemmer, so for that code you can look here and extend things as needed. Jeez, I really need to organise things. – cs95 May 31, 2018 at 14:43

1 @coldspeed, So, I do have a question. How would you include ALL non alphabetical characters in `punct`? Something like `re.compile(r"[^a-zA-Z]")`. I process a lot of text with special characters like ™ and ˚ etc. so I need to get rid of all that crap. I think including them in `punct` explicitly would be too much work as there's too many characters (and I noticed `str.maketrans` doesn't pick up on all these special characters) – PyRsquared Jun 1, 2018 at 10:04

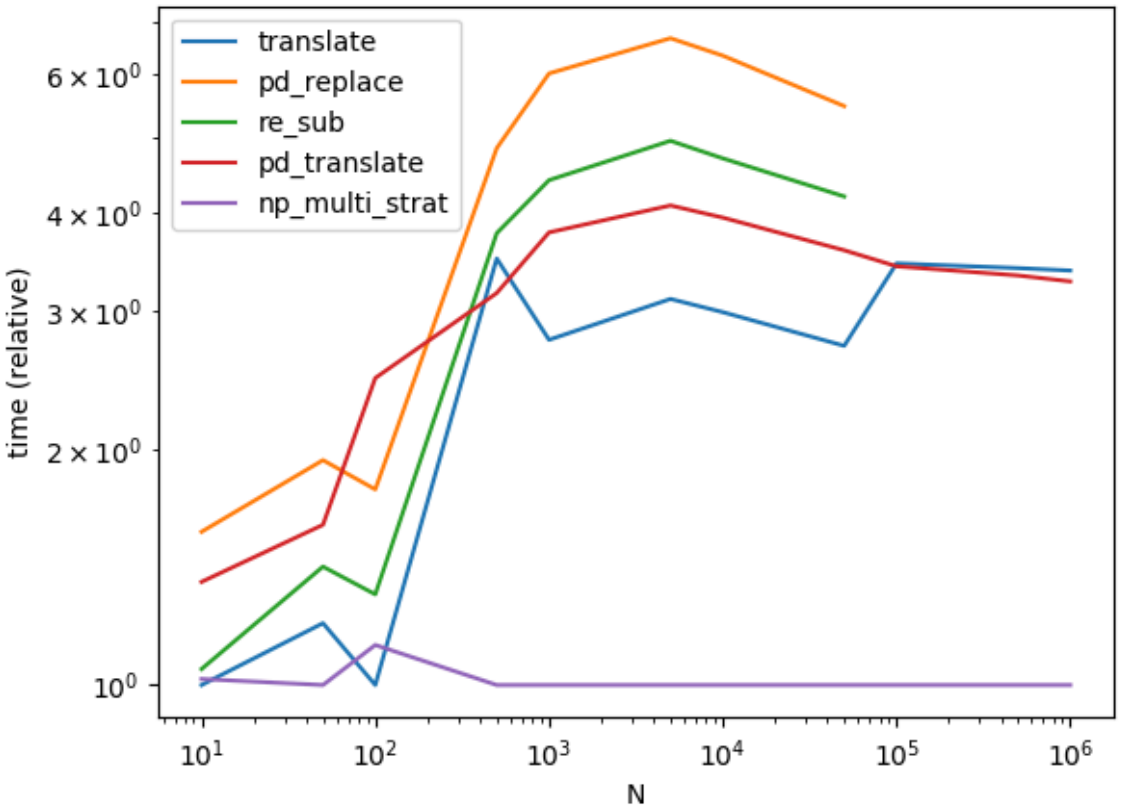
1 That's the smallest value range I've ever seen a log scale used for, assuming that is a log scale on the vertical axis of that graph. – user2357112 Jun 22, 2018 at 18:32

37

Using numpy we can gain a healthy speedup over the best methods posted so far. The basic strategy is similar---make one big super string. But the processing seems much faster in numpy, presumably because we fully exploit the simplicity of the nothing-for-something replacement op.

For smaller (less than `0x110000` characters total) problems we automatically find a separator, for larger problems we use a slower method that does not rely on `str.split`.

Note that I have moved all precomputables out of the functions. Also note, that `translate` and `pd_translate` get to know the only possible separator for the three largest problems for free whereas `np_multi_strat` has to compute it or to fall back to the separator-less strategy. And finally, note that for the last three data points I switch to a more "interesting" problem; `pd_replace` and `re_sub` because they are not equivalent to the other methods had to be excluded for that.



On the algorithm:

The basic strategy is actually quite simple. There are only `0x110000` different unicode characters. As OP frames the challenge in terms of huge data sets, it is perfectly worthwhile making a lookup table that has `True` at the character id's that we want to keep and `False` at the ones that have to go --- the punctuation in our example.

Such a lookup table can be used for bulk loookup using numpy's advanced indexing. As lookup is fully vectorized and essentially amounts to dereferencing an array of pointers it is much faster than for example dictionary lookup. Here we make use of numpy view casting which allows to reinterpret unicode characters as integers essentially for free.

Using the data array which contains just one monster string reinterpreted as a sequence of numbers to index into the lookup table results in a boolean mask. This mask can then be used to filter out the unwanted characters. Using boolean indexing this, too, is a single line of code.

So far so simple. The tricky bit is chopping up the monster string back into its parts. If we have a separator, i.e. one character that does not occur in the data or the punctuation list, then it still is easy. Use this character to join and resplit. However, automatically finding a separator is challenging and indeed accounts for half the loc in the implementation below.

Alternatively, we can keep the split points in a separate data structure, track how they move as a consequence of deleting unwanted characters and then use them to slice the processed monster string. As chopping up into parts of uneven length is not numpy's strongest suit, this method is slower than `str.split` and only used as a fallback when a separator would be too expensive to calculate if it existed in the first place.

Code (timing/plotting heavily based on @COLDSPEED's post):

```
import numpy as np
import pandas as pd
import string
import re


spct = np.array([string.punctuation]).view(np.int32)
lookup = np.zeros((0x110000,), dtype=bool)
lookup[spct] = True
invlookup = ~lookup
OSEP = spct[0]
SEP = chr(OSEP)
while SEP in string.punctuation:
    OSEP = np.random.randint(0, 0x110000)
    SEP = chr(OSEP)


def find_sep_2(letters):
    letters = np.array([letters]).view(np.int32)
    msk = invlookup.copy()
    msk[letters] = False
    sep = msk.argmax()
    if not msk[sep]:
        return None
    return sep

def find_sep(letters, sep=0x88000):
    letters = np.array([letters]).view(np.int32)
    cmp = np.sign(sep-letters)
    cmpf = np.sign(sep-spct)
    if cmp.sum() + cmpf.sum() >= 1:
        left, right, gs = sep+1, 0x110000, -1
    else:
        left, right, gs = 0, sep, 1
    idx, = np.where(cmp == gs)
    idxf, = np.where(cmpf == gs)
    sep = (left + right) // 2
    while True:
        cmp = np.sign(sep-letters[idx])
        cmpf = np.sign(sep-spct[idxf])
        if cmp.all() and cmpf.all():
            return sep
        if cmp.sum() + cmpf.sum() >= (left & 1 == right & 1):
            left, sep, gs = sep+1, (right + sep) // 2, -1
        else:
            right, sep, gs = sep, (left + sep) // 2, 1
        idx = idx[cmp == gs]
        idxf = idxf[cmpf == gs]
```

```
def np_multi_strat(df):
    L = df['text'].tolist()
    all_ = ''.join(L)
    sep = 0x088000
    if chr(sep) in all_: # very unlikely ...
        if len(all_) >= 0x110000: # fall back to separator-less method
                                  # (finding separator too expensive)
            LL = np.array((0, *map(len, L)))
            LLL = LL.cumsum()
            all_ = np.array([all_]).view(np.int32)
            pnct = invlookup[all_]
            NL = np.add.reduceat(pnct, LLL[:-1])
            NLL = np.concatenate([[0], NL.cumsum()]).tolist()
            all_ = all_[pnct]
            all_ = all_.view(f'U{all_.size}').item(0)
            return df.assign(text=[all_[NLL[i]:NLL[i+1]]
                                   for i in range(len(NLL)-1)])
        elif len(all_) >= 0x22000: # use mask
            sep = find_sep_2(all_)
        else: # use bisection
            sep = find_sep(all_)
    all_ = np.array([chr(sep).join(L)]).view(np.int32)
    pnct = invlookup[all_]
    all_ = all_[pnct]
    all_ = all_.view(f'U{all_.size}').item(0)
    return df.assign(text=all_.split(chr(sep)))

def pd_replace(df):
    return df.assign(text=df['text'].str.replace(r'[^\w\s]+', ''))


p = re.compile(r'[^\w\s]+')

def re_sub(df):
    return df.assign(text=[p.sub('', x) for x in df['text'].tolist()])

punct = string.punctuation.replace(SEP, '')
transtab = str.maketrans(dict.fromkeys(punct, ''))

def translate(df):
    return df.assign(
        text=SEP.join(df['text'].tolist()).translate(transtab).split(SEP)
    )

# MaxU's version (https://stackoverflow.com/a/50444659/4909087)
def pd_translate(df):
    return df.assign(text=df['text'].str.translate(transtab))

from timeit import timeit

import pandas as pd
import matplotlib.pyplot as plt

res = pd.DataFrame(
       index=['translate', 'pd_replace', 're_sub', 'pd_translate',
'np_multi_strat'],
       columns=[10, 50, 100, 500, 1000, 5000, 10000, 50000, 100000, 500000,
                1000000],
       dtype=float
)

for c in res.columns:
```

```
    if c >= 100000: # stress test the separator finder
        all_ = np.r_[:OSEP, OSEP+1:0x110000].repeat(c//10000)
        np.random.shuffle(all_)
        split = np.arange(c-1) + \
                np.sort(np.random.randint(0, len(all_) - c + 2, (c-1,)))
        l = [x.view(f'U{x.size}').item(0) for x in np.split(all_, split)]
    else:
        l = ['a..b?!??', '%hgh&12','abc123!!!', '$$$1234'] * c
    df = pd.DataFrame({'text' : l})
    for f in res.index:
        if f == res.index[0]:
            ref = globals()[f](df).text
        elif not (ref == globals()[f](df).text).all():
            res.at[f, c] = np.nan
            print(f, 'disagrees at', c)
            continue
        stmt = '{}(df)'.format(f)
        setp = 'from __main__ import df, {}'.format(f)
        res.at[f, c] = timeit(stmt, setp, number=16)

ax = res.div(res.min()).T.plot(loglog=True)
ax.set_xlabel("N");
ax.set_ylabel("time (relative)");

plt.show()
```

Share Follow

edited May 25, 2018 at 0:16

answered May 24, 2018 at 22:07

Paul Panzer

53k ● 3 ● 58 ● 103

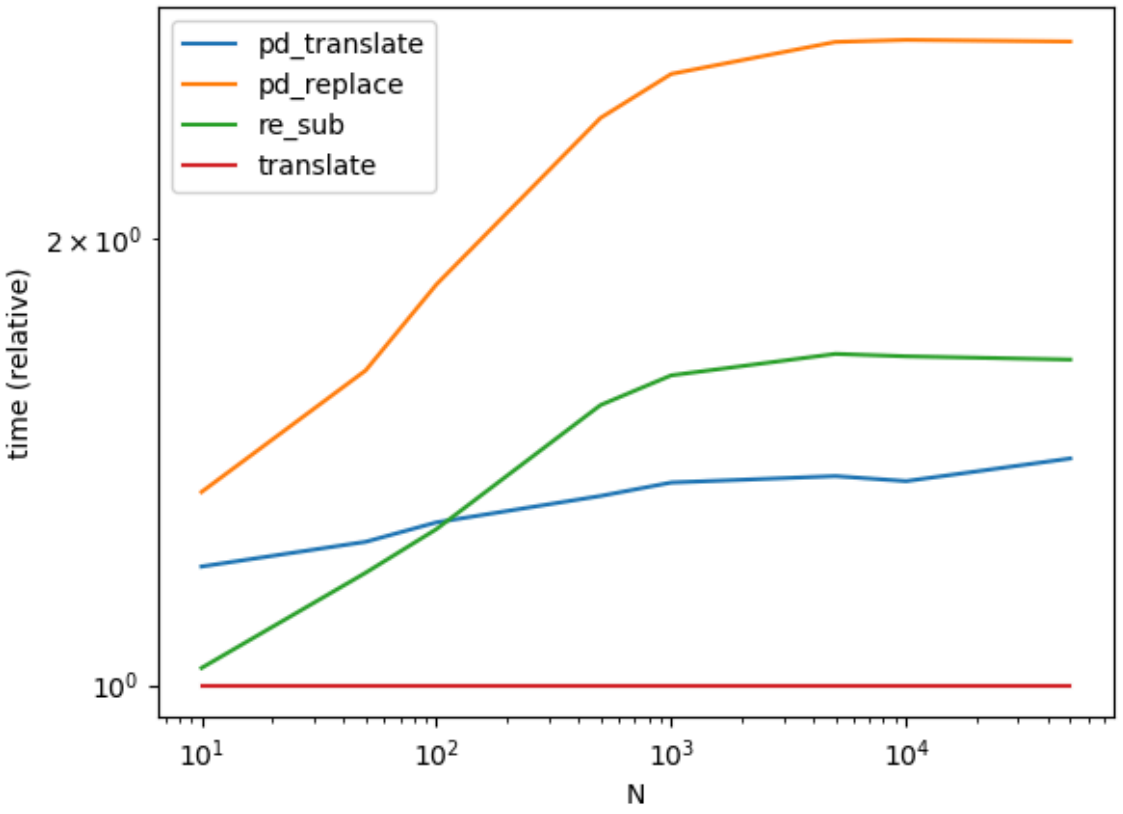
2 I love this answer and the appreciate amount of work that has gone into it. This certainly challenges the limits of performance for operations like this as we know it. A couple of minor remarks, 1) can you explain/document your code so it is a little more clear what certain subroutines are doing? 2) at low values of N the overhead essentially outweighs the performance, and 3) I'd be interested to see how this compares in terms of memory. Overall, awesome job! – cs95 May 24, 2018 at 23:07

1 @coldspeed 1) I've given it a try. Hope it helps. 2) Yep, that's numpy for you. 3) Memory might be a problem because we are creating the superstring, then numpyfy it which creates a copy then create the mask of the same dimensions then filter which creates another copy. – Paul Panzer May 25, 2018 at 0:12

20

Interesting enough that vectorized Series.str.translate method is still slightly slower compared to Vanilla Python `str.translate()`:

```
def pd_translate(df):
    return df.assign(text=df['text'].str.translate(transtab))
```



Share Follow

answered May 21, 2018 at 8:20

MaxU - stand with Ukraine

211k ● 37 ● 401 ● 432

I assume the reason is because we are performing N translations instead of joining, doing one, and splitting. – cs95 May 21, 2018 at 8:21

@coldspeed, yeah, i think so too – MaxU - stand with Ukraine May 21, 2018 at 8:22

2 try this with a NaN and see what happens – Jeff May 25, 2018 at 1:20

0

Using a user defined function and `apply`. I needed to make a custom list of punctuations, so I used the below code and replaced my list of punctuations with `string.punctuation`

```
import string
def remove_punctuation(text):
    return text.translate(str.maketrans('', '', string.punctuation))

df["new_col_name"] = df["col_name"].apply(remove_punctuation)
```

Share Follow

answered Jul 27, 2022 at 17:28

user3503711
2,036 ● 1 ● 30 ● 37

Highly active question. Earn 10 reputation (not counting the association bonus) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.