Change column type in pandas

Asked 9 years, 7 months ago Modified 2 months ago Viewed 3.2m times

1374

I created a DataFrame from a list of lists:

```
table = [
    ['a',  '1.2',  '4.2' ],
    ['b',  '70',   '0.03'],
    ['x',  '5',    '0'   ],
]

df = pd.DataFrame(table)
```

How do I convert the columns to specific types? In this case, I want to convert columns 2 and 3 into floats.

Is there a way to specify the types while converting to DataFrame? Or is it better to create the DataFrame first and then loop through the columns to change the type for each column? Ideally I would like to do this in a dynamic way because there can be hundreds of columns, and I don't want to specify exactly which columns are of which type. All I can guarantee is that each column contains values of the same type.

python pandas dataframe types casting

Share Edit Follow

edited Aug 23 at 15:31
tdy
30.7k 11 63 61

asked Apr 8, 2013 at 23:53
user1642513

14 Answers

Sorted by: Highest score (default)

2279

+200

You have four main options for converting types in pandas:

1. `to_numeric()` - provides functionality to safely convert non-numeric types (e.g. strings) to a suitable numeric type. (See also `to_datetime()` and `to_timedelta()`.)
2. `astype()` - convert (almost) any type to (almost) any other type (even if it's not necessarily sensible to do so). Also allows you to convert to categorial types (very useful).
3. `infer_objects()` - a utility method to convert object columns holding Python objects to a pandas type if possible.
4. `convert_dtypes()` - convert DataFrame columns to the "best possible" dtype that supports `pd.NA` (pandas' object to indicate a missing value).

Read on for more detailed explanations and usage of each of these methods.

1. `to_numeric()`

The best way to convert one or more columns of a DataFrame to numeric values is to use `pandas.to_numeric()`.

This function will try to change non-numeric objects (such as strings) into integers or floating-point numbers as appropriate.

Basic usage

The input to `to_numeric()` is a Series or a single column of a DataFrame.

```
>>> s = pd.Series(["8", 6, "7.5", 3, "0.9"]) # mixed string and numeric values
>>> s
0      8
1      6
2    7.5
3      3
4    0.9
dtype: object

>>> pd.to_numeric(s) # convert everything to float values
0    8.0
1    6.0
2    7.5
3    3.0
4    0.9
dtype: float64
```

As you can see, a new Series is returned. Remember to assign this output to a variable or column name to continue using it:

```
# convert Series
my_series = pd.to_numeric(my_series)

# convert column "a" of a DataFrame
df["a"] = pd.to_numeric(df["a"])
```

You can also use it to convert multiple columns of a DataFrame via the `apply()` method:

```
# convert all columns of DataFrame
df = df.apply(pd.to_numeric) # convert all columns of DataFrame

# convert just columns "a" and "b"
df[["a", "b"]] = df[["a", "b"]].apply(pd.to_numeric)
```

As long as your values can all be converted, that's probably all you need.

Error handling

But what if some values can't be converted to a numeric type?

`to_numeric()` also takes an `errors` keyword argument that allows you to force non-numeric values to be `NaN`, or simply ignore columns containing these values.

Here's an example using a Series of strings `s` which has the object dtype:

```
>>> s = pd.Series(['1', '2', '4.7', 'pandas', '10'])
>>> s
0         1
1         2
2       4.7
3    pandas
4        10
dtype: object
```

The default behaviour is to raise if it can't convert a value. In this case, it can't cope with the string 'pandas':

```
>>> pd.to_numeric(s) # or pd.to_numeric(s, errors='raise')
ValueError: Unable to parse string
```

Rather than fail, we might want 'pandas' to be considered a missing/bad numeric value. We can coerce invalid values to `NaN` as follows using the `errors` keyword argument:

```
>>> pd.to_numeric(s, errors='coerce')
0     1.0
1     2.0
2     4.7
3     NaN
4    10.0
dtype: float64
```

The third option for `errors` is just to ignore the operation if an invalid value is encountered:

```
>>> pd.to_numeric(s, errors='ignore')
# the original Series is returned untouched
```

This last option is particularly useful for converting your entire DataFrame, but don't know which of our columns can be converted reliably to a numeric type. In that case, just write:

```
df.apply(pd.to_numeric, errors='ignore')
```

The function will be applied to each column of the DataFrame. Columns that can be converted to a numeric type will be converted, while columns that cannot (e.g. they contain non-digit strings or dates) will be left alone.

Downcasting

By default, conversion with `to_numeric()` will give you either an `int64` or `float64` dtype (or whatever integer width is native to your platform).

That's usually what you want, but what if you wanted to save some memory and use a more compact dtype, like `float32`, or `int8`?

`to_numeric()` gives you the option to downcast to either `'integer'`, `'signed'`, `'unsigned'`, `'float'`. Here's an example for a simple series `s` of integer type:

```
>>> s = pd.Series([1, 2, -7])
>>> s
0    1
1    2
2   -7
dtype: int64
```

Downcasting to `'integer'` uses the smallest possible integer that can hold the values:

```
>>> pd.to_numeric(s, downcast='integer')
0    1
1    2
2   -7
dtype: int8
```

Downcasting to `'float'` similarly picks a smaller than normal floating type:

```
>>> pd.to_numeric(s, downcast='float')
0    1.0
1    2.0
2   -7.0
dtype: float32
```

2. `astype()`

The `astype()` method enables you to be explicit about the dtype you want your DataFrame or Series to have. It's very versatile in that you can try and go from one type to any other.

Basic usage

Just pick a type: you can use a NumPy dtype (e.g. `np.int16`), some Python types (e.g. bool), or pandas-specific types (like the categorical dtype).

Call the method on the object you want to convert and `astype()` will try and convert it for you:

```
# convert all DataFrame columns to the int64 dtype
df = df.astype(int)

# convert column "a" to int64 dtype and "b" to complex type
df = df.astype({"a": int, "b": complex})

# convert Series to float16 type
s = s.astype(np.float16)

# convert Series to Python strings
s = s.astype(str)

# convert Series to categorical type - see docs for more details
s = s.astype('category')
```

Notice I said "try" - if `astype()` does not know how to convert a value in the Series or DataFrame, it will raise an error. For example, if you have a `NaN` or `inf` value you'll get an error trying to convert it to an integer.

As of pandas 0.20.0, this error can be suppressed by passing `errors='ignore'`. Your original object will be returned untouched.

Be careful

`astype()` is powerful, but it will sometimes convert values "incorrectly". For example:

```
>>> s = pd.Series([1, 2, -7])
>>> s
0    1
1    2
2   -7
dtype: int64
```

These are small integers, so how about converting to an unsigned 8-bit type to save memory?

```
>>> s.astype(np.uint8)
0      1
1      2
2    249
dtype: uint8
```

The conversion worked, but the -7 was wrapped round to become 249 (i.e. 2^8 - 7)!

Trying to downcast using `pd.to_numeric(s, downcast='unsigned')` instead could help prevent this error.

3. `infer_objects()`

Version 0.21.0 of pandas introduced the method `infer_objects()` for converting columns of a DataFrame that have an object datatype to a more specific type (soft conversions).

For example, here's a DataFrame with two columns of object type. One holds actual integers and the other holds strings representing integers:

```
>>> df = pd.DataFrame({'a': [7, 1, 5], 'b': ['3','2','1']}, dtype='object')
>>> df.dtypes
a    object
b    object
dtype: object
```

Using `infer_objects()`, you can change the type of column 'a' to int64:

```
>>> df = df.infer_objects()
>>> df.dtypes
a     int64
b    object
dtype: object
```

Column 'b' has been left alone since its values were strings, not integers. If you wanted to force both columns to an integer type, you could use `df.astype(int)` instead.

4. `convert_dtypes()`

Version 1.0 and above includes a method `convert_dtypes()` to convert Series and DataFrame columns to the best possible dtype that supports the `pd.NA` missing value.

Here "best possible" means the type most suited to hold the values. For example, this a pandas integer type, if all of the values are integers (or missing values): an object column of Python integer objects are converted to `Int64`, a column of NumPy `int32` values, will become the pandas dtype `Int32`.

With our `object` DataFrame `df`, we get the following result:

```
>>> df.convert_dtypes().dtypes
a     Int64
b    string
dtype: object
```

Since column 'a' held integer values, it was converted to the `Int64` type (which is capable of holding missing values, unlike `int64`).

Column 'b' contained string objects, so was changed to pandas' `string` dtype.

By default, this method will infer the type from object values in each column. We can change this by passing `infer_objects=False`:

```
>>> df.convert_dtypes(infer_objects=False).dtypes
a    object
b    string
dtype: object
```

Now column 'a' remained an object column: pandas knows it can be described as an 'integer' column (internally it ran `infer_dtype`) but didn't infer exactly what dtype of integer it should have so did not convert it. Column 'b' was again converted to 'string' dtype as it was recognised as holding 'string' values.

Share Edit Follow

edited Nov 7, 2021 at 1:40
Henry Ecker
33.1k 18 33 53

answered Feb 21, 2015 at 17:37
Alex Riley
160k 44 253 234

11 Also, unlike .astype(float), this will convert strings to NaNs instead of raising an error – Rob Aug 4, 2015 at 8:24

17 `.convert_objects` is depracated since `0.17` - use `df.to_numeric` instead – Matti Lyra Oct 31, 2015 at 14:28

2 is there a way to `error=coerce` in `astype()` ? – fogx May 7, 2021 at 9:58

@fogx No there is not. You can refer here – Kishore Dec 2, 2021 at 11:20

519

Use this:

```
a = [['a', '1.2', '4.2'], ['b', '70', '0.03'], ['x', '5', '0']]
df = pd.DataFrame(a, columns=['one', 'two', 'three'])
df

Out[16]:
  one  two three
0   a  1.2   4.2
1   b   70  0.03
2   x    5     0

df.dtypes

Out[17]:
one      object
two      object
three    object

df[['two', 'three']] = df[['two', 'three']].astype(float)

df.dtypes

Out[19]:
one       object
two      float64
three    float64
```

Share Edit Follow

edited Sep 19 at 17:10
Peter Mortensen
30.6k 21 104 125

answered Apr 21, 2013 at 18:15
hernamesbarbara
6,620 3 25 25

11 Yes! `pd.DataFrame` has a `dtype` argument that might let you do w/ you're looking for. df = pd.DataFrame(a, columns=['one', 'two', 'three'], dtype=float) In [2]: df.dtypes Out[2]: one object two float64 three float64 dtype: object – hernamesbarbara Dec 9, 2013 at 14:12

23 When I try as suggested, I get a warning `SettingWithCopyWarning: A value is trying to be set on a copy of a slice from a DataFrame. Try using .loc[row_index,col_indexer] = value instead`. This may have been introduced in a newer version of pandas and I don't see anything wrong as a result, but I just wonder what this warning is all about. Any idea? – orange Jun 6, 2014 at 7:34

2 @orange the warning is to alert users to potentially confusing behavior with chained operations, and with pandas returning copies of rather than editing dataframes. see stackoverflow.com/questions/20625582/… and related. – A.Wan Jun 27, 2014 at 1:14

25 That's a good method, but it doesn't work when there are NaN in a column. Have no idea why NaN just cannot stay NaN when casting float to int: `ValueError: Cannot convert NA to integer` – Vitaly Isaev Jan 21, 2015 at 11:25

9 @GillBates yes, in a dictionary. `df = pd.DataFrame(a, columns=['one', 'two', 'three'], dtype={'one': str, 'two': int, 'three': float})` . I'm having a hard time finding the specification for accepted "dtype" values though. A list would be nice (currently I do `dict(enumerate(my_list))`). – FichteFoll Jul 7, 2016 at 2:43

50

This below code will change the datatype of a column.

```
df[['col.name1', 'col.name2'...]] = df[['col.name1',
'col.name2'..]].astype('data_type')
```

In place of the data type, you can give your datatype what you want, like, str, float, int, etc.

Share Edit Follow

edited Sep 19 at 17:05
Peter Mortensen
30.6k 21 104 125

answered Nov 15, 2017 at 9:38
Akash Nayak
773 9 13

Mind you that when applying this on a column containing the strings ``` 'True' ``` and ``` 'False' ``` using the data_type `bool` , everything is changed to `True` . – H. Vabri Jan 6, 2018 at 16:28

This option you can also convert to type "category" – neves Sep 22, 2018 at 18:21

32

When I've only needed to specify specific columns, and I want to be explicit, I've used (per *pandas.DataFrame.astype*):

```
dataframe = dataframe.astype({'col_name_1':'int','col_name_2':'float64', etc.
...})
```

So, using the original question, but providing column names to it...

```
a = [['a', '1.2', '4.2'], ['b', '70', '0.03'], ['x', '5', '0']]
df = pd.DataFrame(a, columns=['col_name_1', 'col_name_2', 'col_name_3'])
df = df.astype({'col_name_2':'float64', 'col_name_3':'float64'})
```

Share Edit Follow

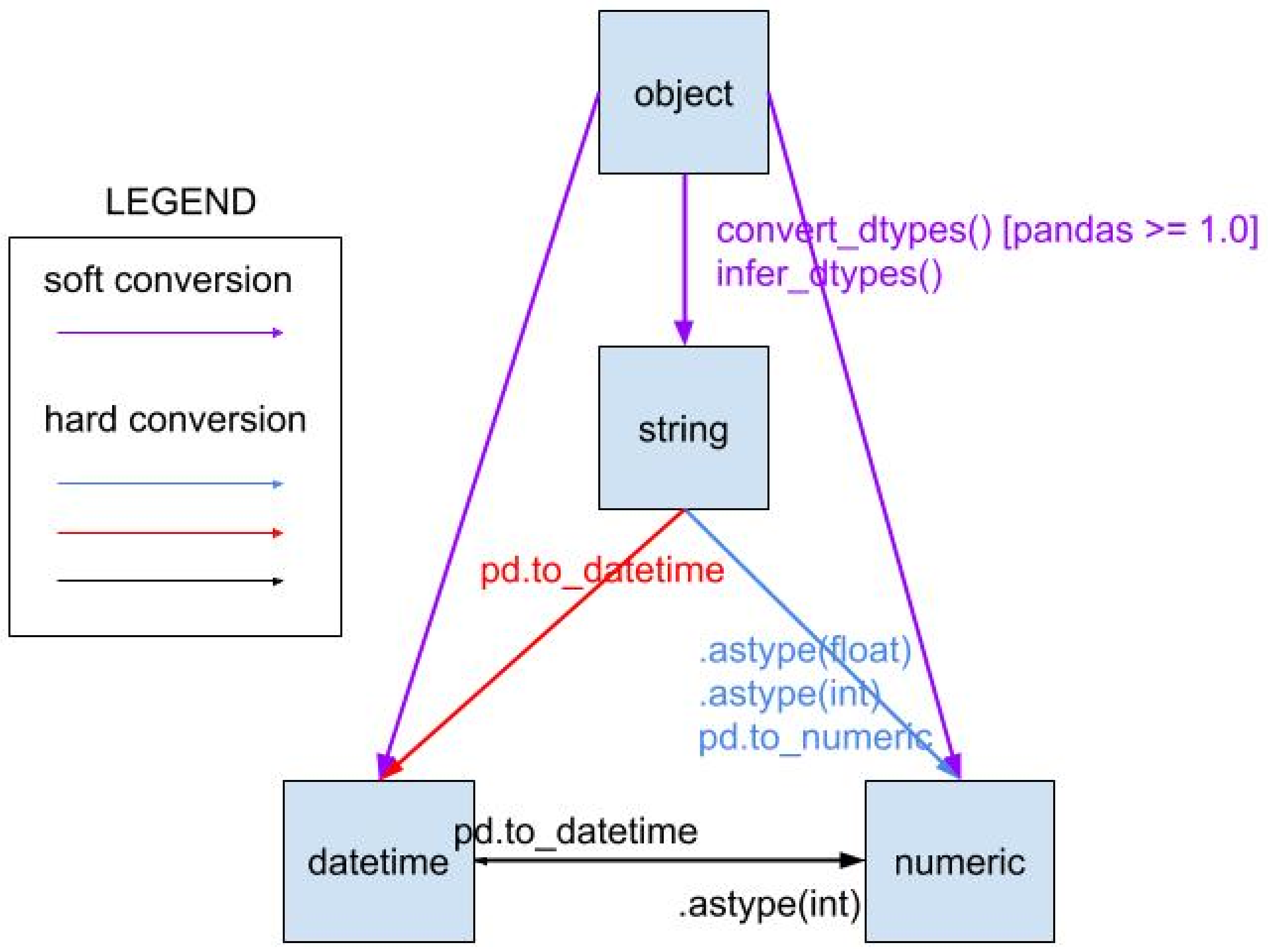
edited Sep 19 at 17:06
Peter Mortensen
30.6k 21 104 125

answered Oct 12, 2018 at 21:02
Thom Ives
3,394 3 29 27

22

pandas >= 1.0

Here's a chart that summarises some of the most important conversions in pandas.



Conversions to string are trivial `.astype(str)` and are not shown in the figure.

"Hard" versus "Soft" conversions

Note that "conversions" in this context could either refer to converting text data into their actual data type (hard conversion), or inferring more appropriate data types for data in object columns (soft conversion). To illustrate the difference, take a look at

```
df = pd.DataFrame({'a': ['1', '2', '3'], 'b': [4, 5, 6]}, dtype=object)
df.dtypes

a    object
b    object
dtype: object

# Actually converts string to numeric - hard conversion
df.apply(pd.to_numeric).dtypes

a    int64
b    int64
dtype: object
```

```
# Infers better data types for object data - soft conversion
df.infer_objects().dtypes

a    object  # no change
b     int64
dtype: object

# Same as infer_objects, but converts to equivalent ExtensionType
    df.convert_dtypes().dtypes
```

Share Edit Follow

edited Sep 19 at 17:41
Peter Mortensen
30.6k 21 104 125

answered Feb 18, 2020 at 10:12
cs95
354k 88 653 705

17

Here is a function that takes as its arguments a DataFrame and a list of columns and coerces all data in the columns to numbers.

```
# df is the DataFrame, and column_list is a list of columns as strings (e.g
["col1","col2","col3"])
# dependencies: pandas

def coerce_df_columns_to_numeric(df, column_list):
    df[column_list] = df[column_list].apply(pd.to_numeric, errors='coerce')
```

So, for your example:

```
import pandas as pd

def coerce_df_columns_to_numeric(df, column_list):
    df[column_list] = df[column_list].apply(pd.to_numeric, errors='coerce')

a = [['a', '1.2', '4.2'], ['b', '70', '0.03'], ['x', '5', '0']]
df = pd.DataFrame(a, columns=['col1','col2','col3'])

coerce_df_columns_to_numeric(df, ['col2','col3'])
```

Share Edit Follow

answered Jun 14, 2017 at 5:42
Harry Stevens
1,313 1 14 18

what if you wanted to use column indexes instead of column names? – jvalenti Jul 5, 2019 at 20:52

15

```
df = df.astype({"columnname": str})
```

#e.g - for changing the column type to string #df is your dataframe

Share Edit Follow

edited Jul 2 at 7:43

answered Sep 27, 2021 at 10:34
Rajeshkanna Purushothaman
630 4 10

2 This duplicate has been flagged to a moderator, as per Flag Duplicate Answers on the same Question. While this is an answer, it duplicates code in the accepted answer and other answers. There is no additional value for SO to keep many answers with the same solution and there doesn't need to be an example for every `type`. Instead, upvote existing answers. – Trenton McKinney Oct 6, 2021 at 0:36

11

Create two dataframes, each with different data types for their columns, and then appending them together:

```
d1 = pd.DataFrame(columns=[ 'float_column' ], dtype=float)
d1 = d1.append(pd.DataFrame(columns=[ 'string_column' ], dtype=str))
```

Results

```
In[8}:  d1.dtypes
Out[8]:
float_column     float64
string_column     object
dtype: object
```

After the dataframe is created, you can populate it with floating point variables in the 1st column, and strings (or any data type you desire) in the 2nd column.

Share Edit Follow

edited Sep 19 at 17:11
Peter Mortensen
30.6k 21 104 125

answered Jul 11, 2017 at 5:56
MikeyE
1,602 1 18 34

9

df.info() gives us initial datatype of temp which is float64

```
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   date    132 non-null    object
 1   temp    132 non-null    float64
```

Now, use this code to change the datatype to int64:

```
df['temp'] = df['temp'].astype('int64')
```

if you do df.info() again, you will see:

```
  #   Column  Non-Null Count  Dtype
 ---  ------  --------------  -----
  0   date    132 non-null    object
  1   temp    132 non-null    int64
```

This shows you have successfully changed the datatype of column temp. Happy coding!

Share Edit Follow

edited Sep 10, 2021 at 11:13
kristianp
5,169 35 56

answered Mar 24, 2021 at 10:15
Mustapha Babatunde
956 9 9

1 I like how df.info() provides memory usage in the final line. – BSalita May 8, 2021 at 8:26

4

Starting pandas 1.0.0, we have `pandas.DataFrame.convert_dtypes`. You can even control what types to convert!

```
In [40]: df = pd.DataFrame(
    ...:     {
    ...:         "a": pd.Series([1, 2, 3], dtype=np.dtype("int32")),
    ...:         "b": pd.Series(["x", "y", "z"], dtype=np.dtype("O")),
    ...:         "c": pd.Series([True, False, np.nan], dtype=np.dtype("O")),
    ...:         "d": pd.Series(["h", "i", np.nan], dtype=np.dtype("O")),
    ...:         "e": pd.Series([10, np.nan, 20], dtype=np.dtype("float")),
    ...:         "f": pd.Series([np.nan, 100.5, 200], dtype=np.dtype("float")),
    ...:     }
    ...: )

In [41]: dff = df.copy()

In [42]: df
Out[42]:
   a  b      c    d     e      f
0  1  x   True    h  10.0    NaN
1  2  y  False    i   NaN  100.5
2  3  z    NaN  NaN  20.0  200.0

In [43]: df.dtypes
Out[43]:
a      int32
b     object
c     object
d     object
e    float64
f    float64
dtype: object

In [44]: df = df.convert_dtypes()

In [45]: df.dtypes
Out[45]:
a      Int32
b     string
c    boolean
d     string
e      Int64
f    float64
dtype: object

In [46]: dff = dff.convert_dtypes(convert_boolean = False)

In [47]: dff.dtypes
Out[47]:
a      Int32
b     string
c     object
d     string
e      Int64
f    float64
dtype: object
```

Share Edit Follow

answered Apr 6, 2020 at 14:26

Sohail
301 4 13

2

> Is there a way to specify the types while converting to DataFrame?

Yes. The other answers convert the dtypes after creating the DataFrame, but we can specify the types at creation. Use either `DataFrame.from_records` or `read_csv(dtype=...)` depending on the input format.

The latter is sometimes necessary to avoid memory errors with big data.

1. `DataFrame.from_records`

Create the DataFrame from a structured array of the desired column types:

```
x = [['foo', '1.2', '70'], ['bar', '4.2', '5']]

df = pd.DataFrame.from_records(np.array(
    [tuple(row) for row in x], # pass a list-of-tuples (x can be a list-of-
lists or 2D array)
    'object, float, int'       # define the column types
))
```

Output:

```
>>> df.dtypes
# f0     object
# f1    float64
# f2      int64
# dtype: object
```

2. `read_csv(dtype=...)`

If you're reading the data from a file, use the `dtype` parameter of `read_csv` to set the column types at load time.

For example, here we read 30M rows with `rating` as 8-bit integers and `genre` as categorical:

```
lines = '''
foo,biography,5
bar,crime,4
baz,fantasy,3
qux,history,2
quux,horror,1
'''
columns = ['name', 'genre', 'rating']
csv = io.StringIO(lines * 6_000_000) # 30M lines

df = pd.read_csv(csv, names=columns, dtype={'rating': 'int8', 'genre':
'category'})
```

In this case, we halve the memory usage upon load:

```
>>> df.info(memory_usage='deep')
# memory usage: 1.8 GB
```

```
>>> pd.read_csv(io.StringIO(lines * 6_000_000)).info(memory_usage='deep')
# memory usage: 3.7 GB
```

This is one way to avoid memory errors with big data. It's not always possible to change the dtypes *after* loading since we might not have enough memory to load the default-typed data in the first place.

Share Edit Follow

answered Mar 16 at 22:03

tdy

30.7k 11 63 61

2

In case you have various objects columns like this Dataframe of 74 Objects columns and 2 Int columns where each value have letters representing units:

```
import pandas as pd
import numpy as np

dataurl =
'https://raw.githubusercontent.com/RubenGavidia/Pandas_Portfolio.py/main/Wes_Mckir

nutrition = pd.read_csv(dataurl,index_col=[0])
nutrition.head(3)
```

Output:

```
    name    serving_size    calories    total_fat    saturated_fat
cholesterol    sodium    choline    folate    folic_acid    ...    fat
saturated_fatty_acids    monounsaturated_fatty_acids
polyunsaturated_fatty_acids    fatty_acids_total_trans    alcohol    ash
caffeine    theobromine    water
0    Cornstarch    100 g    381    0.1g    NaN    0    9.00 mg    0.4 mg
0.00 mcg    0.00 mcg    ...    0.05 g    0.009 g    0.016 g    0.025 g    0.00
mg    0.0 g    0.09 g    0.00 mg    0.00 mg    8.32 g
1    Nuts, pecans    100 g    691    72g    6.2g    0    0.00 mg    40.5 mg
22.00 mcg    0.00 mcg    ...    71.97 g    6.180 g    40.801 g    21.614 g
0.00 mg    0.0 g    1.49 g    0.00 mg    0.00 mg    3.52 g
2    Eggplant, raw    100 g    25    0.2g    NaN    0    2.00 mg    6.9 mg
22.00 mcg    0.00 mcg    ...    0.18 g    0.034 g    0.016 g    0.076 g    0.00
mg    0.0 g    0.66 g    0.00 mg    0.00 mg    92.30 g
3 rows × 76 columns

nutrition.dtypes
name             object
serving_size     object
calories          int64
total_fat        object
saturated_fat    object
                  ...
alcohol          object
ash              object
caffeine         object
theobromine      object
water            object
Length: 76, dtype: object

nutrition.dtypes.value_counts()
object    74
int64      2
dtype: int64
```

A good way to convert to numeric all columns is using regular expressions to replace the units for nothing and astype(float) for change the columns data type to float:

```
nutrition.index = pd.RangeIndex(start = 0, stop = 8789, step= 1)
nutrition.set_index('name',inplace = True)
nutrition.replace('[a-zA-Z]','', regex= True, inplace=True)
nutrition=nutrition.astype(float)
nutrition.head(3)
```

Output:

```
serving_size    calories    total_fat    saturated_fat    cholesterol    sodium
choline    folate    folic_acid    niacin    ...    fat
saturated_fatty_acids    monounsaturated_fatty_acids
polyunsaturated_fatty_acids    fatty_acids_total_trans    alcohol    ash
caffeine    theobromine    water
name
Cornstarch    100.0    381.0    0.1    NaN    0.0    9.0    0.4    0.0    0.0
0.000    ...    0.05    0.009    0.016    0.025    0.0    0.0    0.09    0.0
0.0    8.32
Nuts, pecans    100.0    691.0    72.0    6.2    0.0    0.0    40.5    22.0
0.0    1.167    ...    71.97    6.180    40.801    21.614    0.0    0.0    1.49
0.0    0.0    3.52
Eggplant, raw    100.0    25.0    0.2    NaN    0.0    2.0    6.9    22.0
0.0    0.649    ...    0.18    0.034    0.016    0.076    0.0    0.0    0.66
0.0    0.0    92.30
3 rows × 75 columns

nutrition.dtypes
serving_size     float64
calories         float64
total_fat        float64
saturated_fat    float64
cholesterol      float64
                  ...
alcohol          float64
ash              float64
caffeine         float64
theobromine      float64
water            float64
Length: 75, dtype: object

nutrition.dtypes.value_counts()
float64    75
dtype: int64
```

Now the dataset is clean and you are able to do numeric operations with this Dataframe only with regex and astype().

If you want to collect the units and paste on the headers like `cholesterol_mg` you can use this code:

```
nutrition.index = pd.RangeIndex(start = 0, stop = 8789, step= 1)
nutrition.set_index('name',inplace = True)
nutrition.astype(str).replace('[^a-zA-Z]','', regex= True)
units = nutrition.astype(str).replace('[^a-zA-Z]','', regex= True)
units = units.mode()
units = units.replace('', np.nan).dropna(axis=1)
mapper = { k: k + "_" + units[k].at[0] for k in units}
nutrition.rename(columns=mapper, inplace=True)
nutrition.replace('[a-zA-Z]','', regex= True, inplace=True)
nutrition=nutrition.astype(float)
```

Share Edit Follow

edited Sep 19 at 17:44
Peter Mortensen
30.6k 21 104 125

answered Jan 28 at 0:52
rubengavidia0x
401 1 4 17

1

I thought I had the same problem, but actually I have a slight difference that makes the problem easier to solve. For others looking at this question, it's worth checking the format of your input list. In my case the numbers are initially floats, not strings as in the question:

```
a = [['a', 1.2, 4.2], ['b', 70, 0.03], ['x', 5, 0]]
```

But by processing the list too much before creating the dataframe, I lose the types and everything becomes a string.

Creating the data frame via a NumPy array:

```
df = pd.DataFrame(np.array(a))
df

Out[5]:
   0    1     2
0  a  1.2   4.2
1  b   70  0.03
2  x    5     0

df[1].dtype
Out[7]: dtype('O')
```

gives the same data frame as in the question, where the entries in columns 1 and 2 are considered as strings. However doing

```
df = pd.DataFrame(a)

df
Out[10]:
   0     1     2
0  a   1.2  4.20
1  b  70.0  0.03
2  x   5.0  0.00

df[1].dtype
Out[11]: dtype('float64')
```

does actually give a data frame with the columns in the correct format.

Share Edit Follow

edited Sep 19 at 17:08
Peter Mortensen
30.6k 21 104 125

answered Feb 1, 2019 at 9:49
SarahD
91 8

0

I had the same issue.

I could not find any solution that was satisfying. My solution was simply to convert those float into str and remove the '.0' this way.

In my case, I just apply it on the first column:

```
firstCol = list(df.columns)[0]
df[firstCol] = df[firstCol].fillna('').astype(str).apply(lambda x:
x.replace('.0', ''))
```

Share Edit Follow

edited Sep 19 at 17:45
Peter Mortensen
30.6k 21 104 125

answered Jul 6 at 8:21
Laurent T
216 3 7