Pandas Handling Wrong Format

In a real world scenario, data are taken from various sources which causes inconsistencies in format of the data. For example, a column can have data of integer and string type as the data is copied from different sources.

Such inconsistencies can create challenges, making data analysis difficult or even impossible.

Let's look at an example.

```
import pandas as pd

# create dataframe
data = {
    'Country': ['USA', 'Canada', 'Australia', 'Germany', 'Japan'],
    'Date': ['2023-07-20', '2023-07-21', '2023-07-22', '2023-07-23',
'2023-07-24'],
    'Temperature': [25.5, '28.0', 30.2, 22.8, 26.3]
}
df = pd.DataFrame(data)

# calculate the mean temperature
mean_temperature = df['Temperature'].mean()

print(mean_temperature)
```

```
---------------------------------------------------------------------------
-----
TypeError                                 Traceback (most recent call
last)
Cell In[1], line 12
      9 df = pd.DataFrame(data)
     11 # calculate the mean temperature
---> 12 mean_temperature = df['Temperature'].mean()
     14 print(mean_temperature)

File ~\AppData\Local\Programs\Python\Python39\lib\site-packages\
pandas\core\series.py:6221, in Series.mean(self, axis, skipna,
numeric_only, **kwargs)
   6213 @doc(make_doc("mean", ndim=1))
   6214 def mean(
   6215     self,
   (...)
   6219     **kwargs,
   6220 ):
-> 6221     return NDFrame.mean(self, axis, skipna, numeric_only,
**kwargs)
```

```
File ~\AppData\Local\Programs\Python\Python39\lib\site-packages\
pandas\core\generic.py:11978, in NDFrame.mean(self, axis, skipna,
numeric_only, **kwargs)
  11971 def mean(
  11972     self,
  11973     axis: Axis | None = 0,
   (...)
  11976     **kwargs,
  11977 ) -> Series | float:
> 11978     return self._stat_function(
  11979         "mean", nanops.nanmean, axis, skipna, numeric_only,
**kwargs
  11980     )

File ~\AppData\Local\Programs\Python\Python39\lib\site-packages\
pandas\core\generic.py:11935, in NDFrame._stat_function(self, name,
func, axis, skipna, numeric_only, **kwargs)
  11931 nv.validate_func(name, (), kwargs)
  11933 validate_bool_kwarg(skipna, "skipna", none_allowed=False)
> 11935 return self._reduce(
  11936     func, name=name, axis=axis, skipna=skipna,
numeric_only=numeric_only
  11937 )

File ~\AppData\Local\Programs\Python\Python39\lib\site-packages\
pandas\core\series.py:6129, in Series._reduce(self, op, name, axis,
skipna, numeric_only, filter_type, **kwds)
   6124     # GH#47500 - change to TypeError to match other methods
   6125     raise TypeError(
   6126         f"Series.{name} does not allow
{kwd_name}={numeric_only} "
   6127         "with non-numeric dtypes."
   6128     )
-> 6129 return op(delegate, skipna=skipna, **kwds)

File ~\AppData\Local\Programs\Python\Python39\lib\site-packages\
pandas\core\nanops.py:147, in
bottleneck_switch.__call__.<locals>.f(values, axis, skipna, **kwds)
    145         result = alt(values, axis=axis, skipna=skipna, **kwds)
    146 else:
--> 147     result = alt(values, axis=axis, skipna=skipna, **kwds)
    149 return result

File ~\AppData\Local\Programs\Python\Python39\lib\site-packages\
pandas\core\nanops.py:404, in
_datetimelike_compat.<locals>.new_func(values, axis, skipna, mask,
**kwargs)
    401 if datetimelike and mask is None:
    402     mask = isna(values)
--> 404 result = func(values, axis=axis, skipna=skipna, mask=mask,
```

```
**kwargs)
    406 if datetimelike:
    407     result = _wrap_results(result, orig_values.dtype,
fill_value=iNaT)

File ~\AppData\Local\Programs\Python\Python39\lib\site-packages\
pandas\core\nanops.py:719, in nanmean(values, axis, skipna, mask)
    716     dtype_count = dtype
    718 count = _get_counts(values.shape, mask, axis,
dtype=dtype_count)
--> 719 the_sum = values.sum(axis, dtype=dtype_sum)
    720 the_sum = _ensure_numeric(the_sum)
    722 if axis is not None and getattr(the_sum, "ndim", False):

File ~\AppData\Local\Programs\Python\Python39\lib\site-packages\numpy\
core\_methods.py:49, in _sum(a, axis, dtype, out, keepdims, initial,
where)
     47 def _sum(a, axis=None, dtype=None, out=None, keepdims=False,
     48          initial=_NoValue, where=True):
---> 49     return umr_sum(a, axis, dtype, out, keepdims, initial,
where)

TypeError: unsupported operand type(s) for +: 'float' and 'str'
```

Here, the `Temperature` column contains data in an inconsistent format, with a mixture of `float` and `string` types, which is causing a `TypeError`.

With Pandas, we can handle such issues by converting all the values in a column to a specific format.

Convert Data to Correct Format

We can remove inconsistencies in data by converting a column with inconsistencies to a specific format. For example,

```
import pandas as pd

# create dataframe
data = {
    'Country': ['USA', 'Canada', 'Australia', 'Germany', 'Japan'],
    'Date': ['2023-07-20', '2023-07-21', '2023-07-22', '2023-07-23',
'2023-07-24'],
    'Temperature': [25.5, '28.0', 30.2, 22.8, 26.3]
}
df = pd.DataFrame(data)

# convert temperature column to float
df['Temperature'] = df['Temperature'].astype(float)
```

```
# calculate the mean temperature
mean_temperature = df['Temperature'].mean()

print(mean_temperature)
```

```
26.560000000000002
```

In this example, we converted all the values of `Temperature` column to float using `astype()`. This solves the problem of columns with mixed data.

Handling Mixed Date Formats

Another common example of inconsistency in data format that you often encounter in real life is mixed date formats.

Dates can be represented in various formats such as `mm-dd-yyyy`, `dd-mm-yyyy`, `yyyy-mm-dd` etc. Also, different separators such as `/`, `-`, `.` etc can be used.

We can handle this issue by converting the column containing dates to the `DateTime` format.

Let's look at an example.

```
import pandas as pd

# create a sample dataframe with mixed date formats
df = pd.DataFrame({'date': ['2022-12-01', '01/02/2022', '2022-03-23',
'03/02/2022', '3 4 2023', '2023.9.30']})

# convert the date column to datetime format
df['date'] = pd.to_datetime(df['date'], format='mixed', dayfirst=True)
print(df)
```

```
        date
0 2022-12-01
1 2022-01-02
2 2022-03-23
3 2022-03-02
4 2023-03-04
5 2023-09-30
```

In the above example, we converted the mixed date formats to a uniform `yyyy-mm-dd` format. Here,

`format='mixed'`: specifies that the format of each given date can be different
`dayfirst=True`: specifies that the day should be considered before the month when interpreting dates

The `dayfirst` parameter in `pd.to_datetime` is used to specify whether the day comes before the month in your date strings. When `dayfirst=True`, pandas interprets the date string with the day appearing first, followed by the month.

```
import pandas as pd
# Sample data with date strings
data = {'date': ['12/01/2022', '13/02/2022']}
df = pd.DataFrame(data)

# Convert to datetime with dayfirst=True
df['date'] = pd.to_datetime(df['date'], dayfirst=True)
print("With dayfirst=True:\n", df)
```

```
With dayfirst=True:
         date
0 2022-12-01
1 2022-02-13
```

With `dayfirst=True`, `'12/01/2022'` is interpreted as `January 12, 2022`, and `'13/02/2022'` is interpreted as `February 13, 2022`.

```
#Convert to datetime with dayfirst=False
df['date'] = pd.to_datetime(df['date'], dayfirst=False)
print("\nWith dayfirst=False:\n", df)
```

```
With dayfirst=False:
         date
0 2022-01-12
1 2022-02-13
```

Without specifying `dayfirst (or using dayfirst=False)`, '12/01/2022' is interpreted as December 1, 2022, and '13/02/2022' will raise an error in many cases because there is no 13th month.

Key Points

`With dayfirst=True:` Interprets date as DD/MM/YYYY.

`Without dayfirst (default is False):` Interprets date as MM/DD/YYYY if the format is ambiguous.