

# pandas

## Importing pandas

### Getting started and checking your pandas setup

Difficulty: *easy*

1. Import pandas under the name `pd`.

```
In [1]: import pandas as pd
```

2. Print the version of pandas that has been imported.

```
In [2]: pd.__version__
```

```
Out[2]: '1.3.4'
```

3. Print out all the version information of the libraries that are required by the pandas library.

```
In [3]: pd.show_versions()
```

```
INSTALLED VERSIONS
-----
commit      : 945c9ed766a61c7d2c0a7ccb251b6edebf9cb7d5
python       : 3.10.0.final.0
python-bits   : 64
OS          : Windows
OS-release   : 10
Version     : 10.0.19043
machine     : AMD64
processor    : Intel64 Family 6 Model 61 Stepping 4, GenuineIntel
byteorder    : little
LC_ALL       : None
LANG         : None
LOCALE       : English_India.1252

pandas      : 1.3.4
numpy        : 1.21.4
pytz         : 2021.3
dateutil     : 2.8.2
pip          : 21.2.3
setuptools   : 57.4.0
Cython       : None
pytest       : None
hypothesis   : None
sphinx       : None
blosc        : None
feather      : None
xlsxwriter   : None
lxml.etree   : None
html5lib     : None
```

```

pymysql      : None
psycopg2     : None
jinja2       : 3.0.3
IPython      : 7.29.0
pandas_datareader: None
bs4          : None
bottleneck   : None
fsspec        : None
fastparquet   : None
gcsfs         : None
matplotlib   : 3.5.0
numexpr       : None
odfpy         : None
openpyxl     : None
pandas_gbq    : None
pyarrow       : None
pyxlsb        : None
s3fs          : None
scipy         : 1.7.2
sqlalchemy   : None
tables        : None
tabulate     : None
xarray        : None
xlrd          : None
xlwt          : None
numba         : None

```

## DataFrame basics

A few of the fundamental routines for selecting, sorting, adding and aggregating data in DataFrames

Difficulty: *easy*

Note: remember to import numpy using:

```
import numpy as np
```

Consider the following Python dictionary `data` and Python list `labels`:

```

data = {'animal': ['cat', 'cat', 'snake', 'dog', 'dog', 'cat', 'snake', 'cat',
'dog', 'dog'],
       'age': [2.5, 3, 0.5, np.nan, 5, 2, 4.5, np.nan, 7, 3],
       'visits': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],
       'priority': ['yes', 'yes', 'no', 'yes', 'no', 'no', 'yes', 'no',
'no']}
labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']

```

(This is just some meaningless data I made up with the theme of animals and trips to a vet.)

4. Create a DataFrame `df` from this dictionary `data` which has the index `labels`.

In [5]:

```
import numpy as np

data = {'animal': ['cat', 'cat', 'snake', 'dog', 'dog', 'cat', 'snake', 'cat', 'dog', 'dog'],
       'age': [2.5, 3, 0.5, np.nan, 5, 2, 4.5, np.nan, 7, 3],
       'visits': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],
       'priority': ['yes', 'yes', 'no', 'yes', 'no', 'no', 'yes', 'no', 'no']}
```

```
'age': [2.5, 3, 0.5, np.nan, 5, 2, 4.5, np.nan, 7, 3],
'visits': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],
'priority': ['yes', 'yes', 'no', 'yes', 'no', 'no', 'no', 'yes', 'no', 'no']}}

labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']

df = pd.DataFrame(data , index=labels)
print(df)
```

	animal	age	visits	priority
a	cat	2.5	1	yes
b	cat	3.0	3	yes
c	snake	0.5	2	no
d	dog	NaN	3	yes
e	dog	5.0	2	no
f	cat	2.0	3	no
g	snake	4.5	1	no
h	cat	NaN	1	yes
i	dog	7.0	2	no
j	dog	3.0	1	no

5. Display a summary of the basic information about this DataFrame and its data.

In [7]: df.info()

```
<class 'pandas.core.frame.DataFrame'>
Index: 10 entries, a to j
Data columns (total 4 columns):
 #   Column    Non-Null Count  Dtype  
 --- 
  0   animal     10 non-null    object 
  1   age        8 non-null    float64
  2   visits     10 non-null    int64  
  3   priority   10 non-null    object 
dtypes: float64(1), int64(1), object(2)
memory usage: 400.0+ bytes
```

6. Return the first 3 rows of the DataFrame df .

In [8]: df.head(3)

Out[8]:

	animal	age	visits	priority
a	cat	2.5	1	yes
b	cat	3.0	3	yes
c	snake	0.5	2	no

7. Select just the 'animal' and 'age' columns from the DataFrame df .

In [10]:

```
print(df['animal'])
print(df['age'])
```

	animal	age
a	cat	2.5
b	cat	3.0
c	snake	0.5
d	dog	NaN

```

e      dog
f      cat
g    snake
h      cat
i      dog
j      dog
Name: animal, dtype: object
a    2.5
b    3.0
c    0.5
d    NaN
e    5.0
f    2.0
g    4.5
h    NaN
i    7.0
j    3.0
Name: age, dtype: float64

```

**8.** Select the data in rows [3, 4, 8] and in columns ['animal', 'age'] .

```
In [13]: print(df.iloc[[3, 4, 8], [0, 1]])
```

	animal	age
d	dog	NaN
e	dog	5.0
i	dog	7.0

**9.** Select only the rows where the number of visits is greater than 3.

```
In [15]: print(df[df['visits'] > 3])
```

Empty DataFrame  
Columns: [animal, age, visits, priority]  
Index: []

**10.** Select the rows where the age is missing, i.e. is NaN .

```
In [14]: print(df[df['age'].isnull()])
```

	animal	age	visits	priority
d	dog	NaN	3	yes
h	cat	NaN	1	yes

**11.** Select the rows where the animal is a cat and the age is less than 3.

```
In [20]: print(df[(df['animal'] == 'Cat') & (df['age'] < 3)])
```

Empty DataFrame  
Columns: [animal, age, visits, priority]  
Index: []

**12.** Select the rows the age is between 2 and 4 (inclusive).

```
In [21]: print(df[(df['age'] > 2) & (df['age'] < 4)])
```

	animal	age	visits	priority
a	cat	2.5	1	yes

```
b    cat  3.0      3    yes
j    dog  3.0      1    no
```

**13.** Change the age in row 'f' to 1.5.

```
In [24]: df.loc['f', 'age'] = 1.5
print(df)
```

	animal	age	visits	priority
a	cat	2.5	1	yes
b	cat	3.0	3	yes
c	snake	0.5	2	no
d	dog	NaN	3	yes
e	dog	5.0	2	no
f	cat	1.5	3	no
g	snake	4.5	1	no
h	cat	NaN	1	yes
i	dog	7.0	2	no
j	dog	3.0	1	no

**14.** Calculate the sum of all visits (the total number of visits).

```
In [25]: df['visits'].sum()
```

```
Out[25]: 19
```

**15.** Calculate the mean age for each different animal in `df`.

```
In [26]: df['age'].mean()
```

```
Out[26]: 3.375
```

**16.** Append a new row 'k' to `df` with your choice of values for each column. Then delete that row to return the original DataFrame.

```
In [28]: df.loc['k'] = ['Cat', 2.5, 1, 'yes']
print(df)
df = df.drop('k')
print(df)
```

	animal	age	visits	priority
a	cat	2.5	1	yes
b	cat	3.0	3	yes
c	snake	0.5	2	no
d	dog	NaN	3	yes
e	dog	5.0	2	no
f	cat	1.5	3	no
g	snake	4.5	1	no
h	cat	NaN	1	yes
i	dog	7.0	2	no
j	dog	3.0	1	no
k	Cat	2.5	1	yes

	animal	age	visits	priority
a	cat	2.5	1	yes
b	cat	3.0	3	yes
c	snake	0.5	2	no

	animal	age	visits	priority
d	dog	Nan	3	yes
e	dog	5.0	2	no
f	cat	1.5	3	no
g	snake	4.5	1	no
h	cat	Nan	1	yes
i	dog	7.0	2	no
j	dog	3.0	1	no

17. Count the number of each type of animal in df .

```
In [29]: df['animal'].value_counts()
```

```
Out[29]: cat    4
          dog    4
          snake   2
          Name: animal, dtype: int64
```

18. Sort df first by the values in the 'age' in *descending* order, then by the value in the 'visit' column in *ascending* order.

```
In [30]: df.sort_values(by=['age', 'visits'], ascending=[False, True])
```

	animal	age	visits	priority
i	dog	7.0	2	no
e	dog	5.0	2	no
g	snake	4.5	1	no
j	dog	3.0	1	no
b	cat	3.0	3	yes
a	cat	2.5	1	yes
f	cat	1.5	3	no
c	snake	0.5	2	no
h	cat	Nan	1	yes
d	dog	Nan	3	yes

19. The 'priority' column contains the values 'yes' and 'no'. Replace this column with a column of boolean values: 'yes' should be True and 'no' should be False .

```
In [31]: df['priority'] = df['priority'].map({'yes': True, 'no': False})
```

20. In the 'animal' column, change the 'snake' entries to 'python'.

```
In [32]: df['animal'] = df['animal'].replace('snake', 'python')
```

21. For each animal type and each number of visits, find the mean age. In other words, each row is an animal, each column is a number of visits and the values are the mean ages (hint: use a pivot table).

In [33]: `df.pivot_table(index='animal', columns='visits', values='age', aggfunc='mean')`

Out[33]:

	visits	1	2	3
<b>animal</b>				
<b>cat</b>	2.5	NaN	2.25	
<b>dog</b>	3.0	6.0	NaN	
<b>python</b>	4.5	0.5	NaN	

## DataFrames: beyond the basics

**Slightly trickier: you may need to combine two or more methods to get the right answer**

Difficulty: *medium*

The previous section was tour through some basic but essential DataFrame operations. Below are some ways that you might need to cut your data, but for which there is no single "out of the box" method.

**22.** You have a DataFrame `df` with a column 'A' of integers. For example:

```
df = pd.DataFrame({'A': [1, 2, 2, 3, 4, 5, 5, 5, 6, 7, 7]})
```

How do you filter out rows which contain the same integer as the row immediately above?

In [35]:

```
df = pd.DataFrame({'A': [1, 2, 2, 3, 4, 5, 5, 5, 6, 7, 7]})
df.loc[df['A'].shift() != df['A']]
```

Out[35]:

	A
<b>0</b>	1
<b>1</b>	2
<b>3</b>	3
<b>4</b>	4
<b>5</b>	5
<b>8</b>	6
<b>9</b>	7

**23.** Given a DataFrame of numeric values, say

```
df = pd.DataFrame(np.random.random(size=(5, 3))) # a 5x3 frame of float values
```

how do you subtract the row mean from each element in the row?

In [36]:

```
df = pd.DataFrame(np.random.random(size=(5, 3))) # a 5x3 frame of float values
df.sub(df.mean(axis=1), axis=0)
```

Out[36]:

	0	1	2
<b>0</b>	-0.140370	0.241695	-0.101324
<b>1</b>	-0.166906	-0.163346	0.330252
<b>2</b>	0.293759	0.342053	-0.635812
<b>3</b>	0.237844	-0.530534	0.292690
<b>4</b>	0.309542	-0.162635	-0.146907

**24.** Suppose you have DataFrame with 10 columns of real numbers, for example:

```
df = pd.DataFrame(np.random.random(size=(5, 10)), columns=list('abcdefghijkl'))
```

Which column of numbers has the smallest sum? (Find that column's label.)

In [37]:

```
df = pd.DataFrame(np.random.random(size=(5, 10)), columns=list('abcdefghijkl'))
df.sum().idxmin()
```

Out[37]: 'i'

**25.** How do you count how many unique rows a DataFrame has (i.e. ignore all rows that are duplicates)?

In [38]:

```
len(df) - df.duplicated(keep=False).sum()
```

Out[38]: 5

The next three puzzles are slightly harder...

**26.** You have a DataFrame that consists of 10 columns of floating--point numbers. Suppose that exactly 5 entries in each row are NaN values. For each row of the DataFrame, find the *column* which contains the *third* NaN value.

(You should return a Series of column labels.)

In [39]:

```
(df.isnull().cumsum(axis=1) == 3).idxmax(axis=1)
```

Out[39]:

0	a
1	a
2	a
3	a
4	a

dtype: object

**27.** A DataFrame has a column of groups 'grps' and and column of numbers 'vals'. For example:

```
df = pd.DataFrame({'grps': list('aaabbcaabcccbc'),
                   'vals': [12, 345, 3, 1, 45, 14, 4, 52, 54, 23, 235, 21, 57, 3, 87]})
```

For each *group*, find the sum of the three greatest values.

In [42]:

```
df = pd.DataFrame({'grps': list('aaabbcaabcccbc'),
```

```
'vals': [12, 345, 3, 1, 45, 14, 4, 52, 54, 23, 235, 21, 57, 3, 87]})  
df.groupby('grps')['vals'].nlargest(3).sum(level=0)
```

C:\Users\lhris\AppData\Local\Temp\ipykernel\_9292/430217211.py:3: FutureWarning: Using the level keyword in DataFrame and Series aggregations is deprecated and will be removed in a future version. Use groupby instead. df.sum(level=1) should use df.groupby(level=1).sum().

```
df.groupby('grps')['vals'].nlargest(3).sum(level=0)
```

Out[42]:

```
grps  
a    409  
b    156  
c    345  
Name: vals, dtype: int64
```

**28.** A DataFrame has two integer columns 'A' and 'B'. The values in 'A' are between 1 and 100 (inclusive). For each group of 10 consecutive integers in 'A' (i.e. (0, 10], (10, 20], ...), calculate the sum of the corresponding values in column 'B'.

In [43]:

```
df.groupby(pd.cut(df['A'], np.arange(0, 101, 10)))[‘B’].sum()
```

```
-----  
KeyError                                                 Traceback (most recent call last)  
C:\Program Files\Python310\lib\site-packages\pandas\core\indexes\base.py in get_loc(self, key, method, tolerance)  
    3360         try:  
-> 3361             return self._engine.get_loc(casted_key)  
    3362         except KeyError as err:
```

```
C:\Program Files\Python310\lib\site-packages\pandas\_libs\index.pyx in pandas._libs.index.IndexEngine.get_loc()  
C:\Program Files\Python310\lib\site-packages\pandas\_libs\index.pyx in pandas._libs.index.IndexEngine.get_loc()
```

```
pandas\_libs\hashtable_class_helper.pxi in pandas._libs.hashtable.PyObjectHashTable.get_item()  
pandas\_libs\hashtable_class_helper.pxi in pandas._libs.hashtable.PyObjectHashTable.get_item()
```

**KeyError: 'A'**

The above exception was the direct cause of the following exception:

```
KeyError                                                 Traceback (most recent call last)  
~\AppData\Local\Temp\ipykernel_9292/577332570.py in <module>  
----> 1 df.groupby(pd.cut(df['A'], np.arange(0, 101, 10)))[‘B’].sum()
```

```
C:\Program Files\Python310\lib\site-packages\pandas\core\frame.py in __getitem__(self, key)  
    3456         if self.columns.nlevels > 1:  
    3457             return self._getitem_multilevel(key)  
-> 3458         indexer = self.columns.get_loc(key)  
    3459         if is_integer(indexer):  
    3460             indexer = [indexer]
```

```
C:\Program Files\Python310\lib\site-packages\pandas\core\indexes\base.py in get_loc(self, key, method, tolerance)
```

```

3361         return self._engine.get_loc(casted_key)
3362     except KeyError as err:
-> 3363         raise KeyError(key) from err
3364
3365     if is_scalar(key) and isna(key) and not self.hasnans:

```

**KeyError: 'A'**

## DataFrames: harder problems

These might require a bit of thinking outside the box...

...but all are solvable using just the usual pandas/NumPy methods (and so avoid using explicit `for` loops).

Difficulty: *hard*

**29.** Consider a DataFrame `df` where there is an integer column 'X':

```
df = pd.DataFrame({'X': [7, 2, 0, 3, 4, 2, 5, 0, 3, 4]})
```

For each value, count the difference back to the previous zero (or the start of the Series, whichever is closer). These values should therefore be `[1, 2, 0, 1, 2, 3, 4, 0, 1, 2]`. Make this a new column 'Y'.

In [46]:

```

df = pd.DataFrame({'X': [7, 2, 0, 3, 4, 2, 5, 0, 3, 4]})
i_zero = np.r_[-1, (df['X'] == 0)[0]] # indices of zeros
id_x = np.arange(len(df))
df['Y'] = id_x - i_zero[np.searchsorted(i_zero - 1, id_x) - 1]

```

Here's an alternative approach based on a [cookbook recipe](#):

In [ ]:

**30.** Consider a DataFrame containing rows and columns of purely numerical data. Create a list of the row-column index locations of the 3 largest values.

In [ ]:

```
df.unstack().sort_values()[-3:].index.tolist()
```

**31.** Given a DataFrame with a column of group IDs, 'grps', and a column of corresponding integer values, 'vals', replace any negative values in 'vals' with the group mean.

In [ ]:

```

def replace(group):
    mask = group < 0
    group[mask] = group[~mask].mean()
    return group

df.groupby(['grps'])['vals'].transform(replace)

```

**32.** Implement a rolling mean over groups with window size 3, which ignores NaN value. For example consider the following DataFrame:

```
>>> df = pd.DataFrame({'group': list('aabbaabbbbabab'),
                      'value': [1, 2, 3, np.nan, 2, 3,
                                np.nan, 1, 7, 3, np.nan, 8]})

>>> df
   group  value
0       a    1.0
1       a    2.0
2       b    3.0
3       b    NaN
4       a    2.0
5       b    3.0
6       b    NaN
7       b    1.0
8       a    7.0
9       b    3.0
10      a    NaN
11      b    8.0
```

The goal is to compute the Series:

0	1.000000
1	1.500000
2	3.000000
3	3.000000
4	1.666667
5	3.000000
6	3.000000
7	2.000000
8	3.666667
9	2.000000
10	4.500000
11	4.000000

E.g. the first window of size three for group 'b' has values 3.0, NaN and 3.0 and occurs at row index 5. Instead of being NaN the value in the new column at this row index should be 3.0 (just the two non-NaN values are used to compute the mean  $(3+3)/2$ )

In [47]:

```
df = pd.DataFrame({'group': list('aabbaabbbbabab'),
                   'value': [1, 2, 3, np.nan, 2, 3,
                             np.nan, 1, 7, 3, np.nan, 8]})

g1 = df.groupby(['group'])['value']           # group values
g2 = df.fillna(0).groupby(['group'])['value'] # fillna, then group values

s = g2.rolling(3, min_periods=1).sum() / g1.rolling(3, min_periods=1).count() # compute

s.reset_index(level=0, drop=True).sort_index() # drop/sort index
```

Out[47]:

0	1.000000
1	1.500000
2	3.000000
3	3.000000
4	1.666667
5	3.000000

```

6    3.000000
7    2.000000
8    3.666667
9    2.000000
10   4.500000
11   4.000000
Name: value, dtype: float64

```

## Series and DatetimeIndex

### Exercises for creating and manipulating Series with datetime data

Difficulty: *easy/medium*

pandas is fantastic for working with dates and times. These puzzles explore some of this functionality.

**33.** Create a DatetimeIndex that contains each business day of 2015 and use it to index a Series of random numbers. Let's call this Series `s`.

In [49]:

```

dti = pd.date_range(start='2015-01-01', end='2015-12-31', freq='B')
s = pd.Series(np.random.rand(len(dti)), index=dti)
print(s)

```

```

2015-01-01    0.248991
2015-01-02    0.871624
2015-01-05    0.719922
2015-01-06    0.200836
2015-01-07    0.656121
...
2015-12-25    0.588195
2015-12-28    0.457661
2015-12-29    0.911494
2015-12-30    0.697204
2015-12-31    0.024273
Freq: B, Length: 261, dtype: float64

```

**34.** Find the sum of the values in `s` for every Wednesday.

In [50]:

```
print(s[s.index.weekday == 2].sum())
```

28.372248309948517

**35.** For each calendar month in `s`, find the mean of values.

In [51]:

```
print(s.resample('M').mean())
```

```

2015-01-31    0.442426
2015-02-28    0.499248
2015-03-31    0.400686
2015-04-30    0.593479
2015-05-31    0.414854
2015-06-30    0.522472
2015-07-31    0.513408
2015-08-31    0.416800
2015-09-30    0.487831

```

```
2015-10-31    0.485644
2015-11-30    0.549107
2015-12-31    0.440687
Freq: M, dtype: float64
```

**36.** For each group of four consecutive calendar months in `s`, find the date on which the highest value occurred.

In [75]: `s.groupby(pd.TimeGrouper('4M')).idxmax()`

```
-----
AttributeError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_9292/3430091876.py in <module>
----> 1 s.groupby(pd.TimeGrouper('4M')).idxmax()

C:\Program Files\Python310\lib\site-packages\pandas\__init__.py in __getattr__(name)
    242         return _SparseArray
    243
--> 244     raise AttributeError(f"module 'pandas' has no attribute '{name}'")
    245
    246
```

**AttributeError:** module 'pandas' has no attribute 'TimeGrouper'

**37.** Create a DateTimeIndex consisting of the third Thursday in each month for the years 2015 and 2016.

In [53]: `pd.date_range('2015-01-01', '2016-12-31', freq='WOM-3THU')`

Out[53]: `DatetimeIndex(['2015-01-15', '2015-02-19', '2015-03-19', '2015-04-16',
 '2015-05-21', '2015-06-18', '2015-07-16', '2015-08-20',
 '2015-09-17', '2015-10-15', '2015-11-19', '2015-12-17',
 '2016-01-21', '2016-02-18', '2016-03-17', '2016-04-21',
 '2016-05-19', '2016-06-16', '2016-07-21', '2016-08-18',
 '2016-09-15', '2016-10-20', '2016-11-17', '2016-12-15'],
 dtype='datetime64[ns]', freq='WOM-3THU')`

## Cleaning Data

### Making a DataFrame easier to work with

Difficulty: *easy/medium*

It happens all the time: someone gives you data containing malformed strings, Python, lists and missing data. How do you tidy it up so you can get on with the analysis?

Take this monstrosity as the DataFrame to use in the following puzzles:

```
df = pd.DataFrame({'From_To': ['LoNDOn_paris', 'MAdrid_miLAn',
 'londON_StockhOlm',
 'Budapest_PaRis', 'Brussels_londOn'],
 'FlightNumber': [10045, np.nan, 10065, np.nan, 10085],
 'RecentDelays': [[23, 47], [], [24, 43, 87], [13], [67, 32]],
 'Airline': ['KLM(!)', '<Air France> (12)', '(British
```

```
Airways. )',  
        '12. Air France', '"Swiss Air"'])
```

(It's some flight data I made up; it's not meant to be accurate in any way.)

**38.** Some values in the the FlightNumber column are missing. These numbers are meant to increase by 10 with each row so 10055 and 10075 need to be put in place. Fill in these missing numbers and make the column an integer column (instead of a float column).

```
In [56]: df = pd.DataFrame({'From_To': ['LoNDON_paris', 'MAdrid_miLAN', 'londON_StockhOlm',  
                                     'Budapest_PaRis', 'Brussels_londOn'],  
                           'FlightNumber': [10045, np.nan, 10065, np.nan, 10085],  
                           'RecentDelays': [[23, 47], [], [24, 43, 87], [13], [67, 32]],  
                           'Airline': ['KLM(!)', '<Air France> (12)', '(British Airways. )',  
                                      '12. Air France', '"Swiss Air"'])  
  
print(df)  
df['FlightNumber'] = df['FlightNumber'].interpolate().astype(int)  
print(df)
```

	From_To	FlightNumber	RecentDelays	Airline
0	LoNDON_paris	10045.0	[23, 47]	KLM(!)
1	MAdrid_miLAN	NaN	[]	<Air France> (12)
2	londON_StockhOlm	10065.0	[24, 43, 87]	(British Airways. )
3	Budapest_PaRis	NaN	[13]	12. Air France
4	Brussels_londOn	10085.0	[67, 32]	"Swiss Air"

	From_To	FlightNumber	RecentDelays	Airline
0	LoNDON_paris	10045	[23, 47]	KLM(!)
1	MAdrid_miLAN	10055	[]	<Air France> (12)
2	londON_StockhOlm	10065	[24, 43, 87]	(British Airways. )
3	Budapest_PaRis	10075	[13]	12. Air France
4	Brussels_londOn	10085	[67, 32]	"Swiss Air"

**39.** The From\_To column would be better as two separate columns! Split each string on the underscore delimiter \_ to give a new temporary DataFrame with the correct values. Assign the correct column names to this temporary DataFrame.

```
In [57]: temp = df.From_To.str.split('_', expand=True)  
temp.columns = ['From', 'To']
```

**40.** Notice how the capitalisation of the city names is all mixed up in this temporary DataFrame. Standardise the strings so that only the first letter is uppercase (e.g. "londON" should become "London".)

```
In [58]: temp['From'] = temp['From'].str.capitalize()  
temp['To'] = temp['To'].str.capitalize()
```

**41.** Delete the From\_To column from df and attach the temporary DataFrame from the previous questions.

```
In [59]: df = df.drop('From_To', axis=1)  
df = df.join(temp)
```

**42.** In the Airline column, you can see some extra punctuation and symbols have appeared around

the airline names. Pull out just the airline name. E.g. '(British Airways.)' should become 'British Airways' .

```
In [60]: df['Airline'] = df['Airline'].str.extract('([a-zA-Z\s]+)', expand=False).str.strip()
```

**43.** In the RecentDelays column, the values have been entered into the DataFrame as a list. We would like each first value in its own column, each second value in its own column, and so on. If there isn't an Nth value, the value should be NaN.

Expand the Series of lists into a DataFrame named `delays` , rename the columns `delay_1` , `delay_2` , etc. and replace the unwanted RecentDelays column in `df` with `delays` .

```
In [61]: delays = df['RecentDelays'].apply(pd.Series)

delays.columns = [ 'delay_{}'.format(n) for n in range(1, len(delays.columns)+1) ]

df = df.drop('RecentDelays', axis=1).join(delays)
```

The DataFrame should look much better now.

## Using MultiIndexes

### Go beyond flat DataFrames with additional index levels

Difficulty: *medium*

Previous exercises have seen us analysing data from DataFrames equipped with a single index level. However, pandas also gives you the possibility of indexing your data using *multiple* levels. This is very much like adding new dimensions to a Series or a DataFrame. For example, a Series is 1D, but by using a MultiIndex with 2 levels we gain of much the same functionality as a 2D DataFrame.

The set of puzzles below explores how you might use multiple index levels to enhance data analysis.

To warm up, we'll look make a Series with two index levels.

**44.** Given the lists `letters = ['A', 'B', 'C']` and `numbers = list(range(10))` , construct a MultiIndex object from the product of the two lists. Use it to index a Series of random numbers. Call this Series `s` .

```
In [63]: letters = ['A', 'B', 'C']
numbers = list(range(10))

mi = pd.MultiIndex.from_product([letters, numbers])
s = pd.Series(np.random.rand(30), index=mi)
print(s)
```

A	0	0.445668
	1	0.161503
	2	0.267891
	3	0.023672

```

4    0.602804
5    0.561375
6    0.297096
7    0.092647
8    0.495319
9    0.761233
B  0    0.113222
1    0.663154
2    0.057665
3    0.869429
4    0.866091
5    0.541148
6    0.158827
7    0.855555
8    0.656235
9    0.896393
C  0    0.631602
1    0.693224
2    0.184726
3    0.777119
4    0.865126
5    0.046950
6    0.096403
7    0.640868
8    0.300656
9    0.242604
dtype: float64

```

- 45.** Check the index of `s` is lexicographically sorted (this is a necessary property for indexing to work correctly with a MultiIndex).

```
In [65]: print(s.index.is_lexsorted())
```

```
True
```

```
C:\Users\lhris\AppData\Local\Temp\ipykernel_9292/3463388138.py:1: FutureWarning: MultiIndex.is_lexsorted is deprecated as a public function, users should use MultiIndex.is_monomonic_increasing instead.
print(s.index.is_lexsorted())
```

- 46.** Select the labels `1`, `3` and `6` from the second level of the MultiIndexed Series.

```
In [66]: print(s.loc[:, [1, 3, 6]])
```

```

A  1    0.161503
    3    0.023672
    6    0.297096
B  1    0.663154
    3    0.869429
    6    0.158827
C  1    0.693224
    3    0.777119
    6    0.096403
dtype: float64

```

- 47.** Slice the Series `s` ; slice up to label 'B' for the first level and from label 5 onwards for the second level.

```
In [67]: print(s.loc[pd.IndexSlice[:'B', 5:]])
```

```
A    5    0.561375
     6    0.297096
     7    0.092647
     8    0.495319
     9    0.761233
B    5    0.541148
     6    0.158827
     7    0.855555
     8    0.656235
     9    0.896393
dtype: float64
```

- 48.** Sum the values in `s` for each label in the first level (you should have Series giving you a total for labels A, B and C).

In [68]:

```
print(s.sum(level=0))
```

```
A    3.709207
B    5.677719
C    4.479280
dtype: float64
```

```
C:\Users\lhris\AppData\Local\Temp/ipykernel_9292/3209327977.py:1: FutureWarning: Using t
he level keyword in DataFrame and Series aggregations is deprecated and will be removed
in a future version. Use groupby instead. df.sum(level=1) should use df.groupby(level=
1).sum().
```

```
print(s.sum(level=0))
```

- 49.** Suppose that `sum()` (and other methods) did not accept a `level` keyword argument. How else could you perform the equivalent of `s.sum(level=1)` ?

In [69]:

```
print(s.unstack().sum(axis=0))
```

```
0    1.190492
1    1.517881
2    0.510282
3    1.670220
4    2.334020
5    1.149473
6    0.552327
7    1.589071
8    1.452211
9    1.900230
dtype: float64
```

- 50.** Exchange the levels of the MultiIndex so we have an index of the form (letters, numbers). Is this new Series properly lexsorted? If not, sort it.

In [72]:

```
new_s = s.swaplevel(0, 1)

# check
print(new_s.index.is_lexsorted())

# sort
print(new_s.sort_index())
```

False

```
0   A    0.445668
    B    0.113222
    C    0.631602
1   A    0.161503
    B    0.663154
    C    0.693224
2   A    0.267891
    B    0.057665
    C    0.184726
3   A    0.023672
    B    0.869429
    C    0.777119
4   A    0.602804
    B    0.866091
    C    0.865126
5   A    0.561375
    B    0.541148
    C    0.046950
6   A    0.297096
    B    0.158827
    C    0.096403
7   A    0.092647
    B    0.855555
    C    0.640868
8   A    0.495319
    B    0.656235
    C    0.300656
9   A    0.761233
    B    0.896393
    C    0.242604
dtype: float64
C:\Users\lhris\AppData\Local\Temp\ipykernel_9292/378288086.py:4: FutureWarning: MultiIndex.is_lexsorted is deprecated as a public function, users should use MultiIndex.is_monotonic_increasing instead.
  print(new_s.index.is_lexsorted())
```

## Minesweeper

### Generate the numbers for safe squares in a Minesweeper grid

Difficulty: *medium to hard*

If you've ever used an older version of Windows, there's a good chance you've played with [Minesweeper](#). If you're not familiar with the game, imagine a grid of squares: some of these squares conceal a mine. If you click on a mine, you lose instantly. If you click on a safe square, you reveal a number telling you how many mines are found in the squares that are immediately adjacent. The aim of the game is to uncover all squares in the grid that do not contain a mine.

In this section, we'll make a DataFrame that contains the necessary data for a game of Minesweeper: coordinates of the squares, whether the square contains a mine and the number of mines found on adjacent squares.

**51.** Let's suppose we're playing Minesweeper on a 5 by 4 grid, i.e.

```
X = 5
Y = 4
```

To begin, generate a DataFrame `df` with two columns, '`x`' and '`y`' containing every coordinate for this grid. That is, the DataFrame should start:

	x	y
0	0	0
1	0	1
2	0	2

In [74]:

```
import pandas as pd
p = pd.tools.util.cartesian_product([np.arange(X), np.arange(Y)])
df = pd.DataFrame(np.asarray(p).T, columns=['x', 'y'])
print(df)
```

---

**AttributeError** Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel\_9292/1490458401.py in <module>
 1 import pandas as pd
----> 2 p = pd.tools.util.cartesian\_product([np.arange(X), np.arange(Y)])
 3 df = pd.DataFrame(np.asarray(p).T, columns=['x', 'y'])
 4 print(df)

```
C:\Program Files\Python310\lib\site-packages\pandas\__init__.py in __getattr__(name)
  242         return _SparseArray
  243
--> 244     raise AttributeError(f"module 'pandas' has no attribute '{name}'")
  245
  246
```

**AttributeError**: module 'pandas' has no attribute 'tools'

**52.** For this DataFrame `df`, create a new column of zeros (safe) and ones (mine). The probability of a mine occurring at each location should be 0.4.

In [77]:

```
p = pd.tools.util.cartesian_product([np.arange(X), np.arange(Y)])
df = pd.DataFrame(np.asarray(p).T, columns=['x', 'y'])

df['mine'] = np.random.binomial(1, 0.4, X*Y)
```

---

**AttributeError** Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel\_9292/3467283940.py in <module>
----> 1 p = pd.tools.util.cartesian\_product([np.arange(X), np.arange(Y)])
 2 df = pd.DataFrame(np.asarray(p).T, columns=['x', 'y'])
 3
 4 df['mine'] = np.random.binomial(1, 0.4, X\*Y)

```
C:\Program Files\Python310\lib\site-packages\pandas\__init__.py in __getattr__(name)
  242         return _SparseArray
  243
--> 244     raise AttributeError(f"module 'pandas' has no attribute '{name}'")
  245
  246
```

**AttributeError:** module 'pandas' has no attribute 'tools'

**53.** Now create a new column for this DataFrame called 'adjacent'. This column should contain the number of mines found on adjacent squares in the grid.

(E.g. for the first row, which is the entry for the coordinate (0, 0) , count how many mines are found on the coordinates (0, 1) , (1, 0) and (1, 1) .)

In [78]:

```
from scipy.signal import convolve2d
mine_grid = df.pivot_table(columns='x', index='y', values='mine')
counts = convolve2d(mine_grid.astype(complex), np.ones((3, 3)), mode='same').real.astype(int)
df['adjacent'] = (counts - mine_grid).ravel('F')
```

**KeyError**

Traceback (most recent call last)

```
~\AppData\Local\Temp\ipykernel_9292\1209196736.py in <module>
      1 from scipy.signal import convolve2d
----> 2 mine_grid = df.pivot_table(columns='x', index='y', values='mine')
      3 counts = convolve2d(mine_grid.astype(complex), np.ones((3, 3)), mode='same').re
al.astype(int)
      4 df['adjacent'] = (counts - mine_grid).ravel('F')
```

```
C:\Program Files\Python310\lib\site-packages\pandas\core\frame.py in pivot_table(self, v
values, index, columns, aggfunc, fill_value, margins, dropna, margins_name, observed,
sort)
```

```
    7949         from pandas.core.reshape.pivot import pivot_table
    7950
-> 7951         return pivot_table(
    7952             self,
    7953             values=values,
```

```
C:\Program Files\Python310\lib\site-packages\pandas\core\reshape\pivot.py in pivot_table
(data, values, index, columns, aggfunc, fill_value, margins, dropna, margins_name, obser
ved, sort)
    93         return table.__finalize__(data, method="pivot_table")
    94
---> 95     table = __internal_pivot_table(
    96         data,
    97         values,
```

```
C:\Program Files\Python310\lib\site-packages\pandas\core\reshape\pivot.py in __internal_
pivot_table(data, values, index, columns, aggfunc, fill_value, margins, dropna, margins_
name, observed, sort)
    139         for i in values:
    140             if i not in data:
--> 141                 raise KeyError(i)
    142
    143         to_filter = []
```

**KeyError:** 'mine'

**54.** For rows of the DataFrame that contain a mine, set the value in the 'adjacent' column to NaN.

In [ ]:

```
df.loc[df['mine'] == 1, 'adjacent'] = np.nan
```

55. Finally, convert the DataFrame to grid of the adjacent mine counts: columns are the x coordinate, rows are the y coordinate.

In [79]:

```
df.drop('mine', axis=1)\n    .set_index(['y', 'x']).unstack()
```

```
-----\nKeyError Traceback (most recent call last)\n~\AppData\Local\Temp\ipykernel_9292\3196222206.py in <module>\n----> 1 df.drop('mine', axis=1)\n      2     .set_index(['y', 'x']).unstack()\n\nC:\Program Files\Python310\lib\site-packages\pandas\util\_decorators.py in wrapper(*args,\n      **kwargs)\n      309             stacklevel=stacklevel,\n      310         )\n--> 311     return func(*args, **kwargs)\n      312 \n      313     return wrapper\n\nC:\Program Files\Python310\lib\site-packages\pandas\core\frame.py in drop(self, labels,\n      axis, index, columns, level, inplace, errors)\n      4904         weight 1.0 0.8\n      4905     """\n-> 4906     return super().drop(\n      4907         labels=labels,\n      4908         axis=axis,\n\nC:\Program Files\Python310\lib\site-packages\pandas\core\generic.py in drop(self, labels,\n      axis, index, columns, level, inplace, errors)\n      4148         for axis, labels in axes.items():\n      4149             if labels is not None:\n-> 4150                 obj = obj._drop_axis(labels, axis, level=level, errors=errors)\n      4151 \n      4152     if inplace:\n\nC:\Program Files\Python310\lib\site-packages\pandas\core\generic.py in _drop_axis(self,\n      labels, axis, level, errors)\n      4183         new_axis = axis.drop(labels, level=level, errors=errors)\n      4184     else:\n-> 4185         new_axis = axis.drop(labels, errors=errors)\n      4186     result = self.reindex(**{axis_name: new_axis})\n      4187 \n\nC:\Program Files\Python310\lib\site-packages\pandas\core\indexes\base.py in drop(self,\n      labels, errors)\n      6015     if mask.any():\n      6016         if errors != "ignore":\n-> 6017             raise KeyError(f"\{labels[mask]}\ not found in axis")\n      6018     indexer = indexer[~mask]\n      6019     return self.delete(indexer)\n\nKeyError: "[ 'mine' ] not found in axis"
```

## Plotting

### Visualize trends and patterns in data

Difficulty: *medium*

To really get a good understanding of the data contained in your DataFrame, it is often essential to create plots: if you're lucky, trends and anomalies will jump right out at you. This functionality is baked into pandas and the puzzles below explore some of what's possible with the library.

**56.** Pandas is highly integrated with the plotting library matplotlib, and makes plotting DataFrames very user-friendly! Plotting in a notebook environment usually makes use of the following boilerplate:

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.style.use('ggplot')
```

`%matplotlib inline` tells the notebook to show plots inline, instead of creating them in a separate window.

`plt.style.use('ggplot')` is a style theme that most people find agreeable, based upon the styling of R's ggplot package.

For starters, make a scatter plot of this random data, but use black X's instead of the default markers.

```
df = pd.DataFrame({"xs":[1,5,2,8,1], "ys":[4,2,1,9,6]})
```

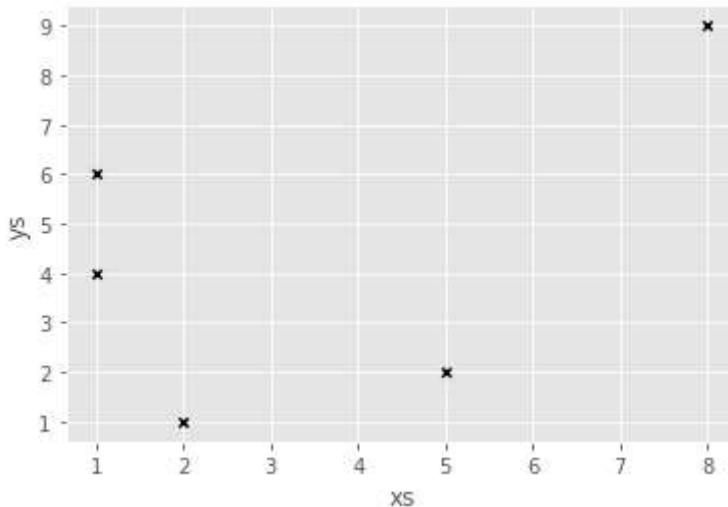
Consult the [documentation](#) if you get stuck!

```
In [80]: import matplotlib.pyplot as plt
%matplotlib inline
plt.style.use('ggplot')

df = pd.DataFrame({"xs":[1,5,2,8,1], "ys":[4,2,1,9,6]})

df.plot.scatter("xs", "ys", color = "black", marker = "x")
```

```
Out[80]: <AxesSubplot:xlabel='xs', ylabel='ys'>
```



**57.** Columns in your DataFrame can also be used to modify colors and sizes. Bill has been keeping track of his performance at work over time, as well as how good he was feeling that day, and whether he had a cup of coffee in the morning. Make a plot which incorporates all four features of this DataFrame.

(Hint: If you're having trouble seeing the plot, try multiplying the Series which you choose to represent size by 10 or more)

*The chart doesn't have to be pretty: this isn't a course in data viz!*

```
df = pd.DataFrame({"productivity": [5, 2, 3, 1, 4, 5, 6, 7, 8, 3, 4, 8, 9],
                   "hours_in"      : [1, 9, 6, 5, 3, 9, 2, 9, 1, 7, 4, 2, 2],
                   "happiness"    : [2, 1, 3, 2, 3, 1, 2, 3, 1, 2, 2, 1, 3],
                   "caffienated"  : [0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0]})
```

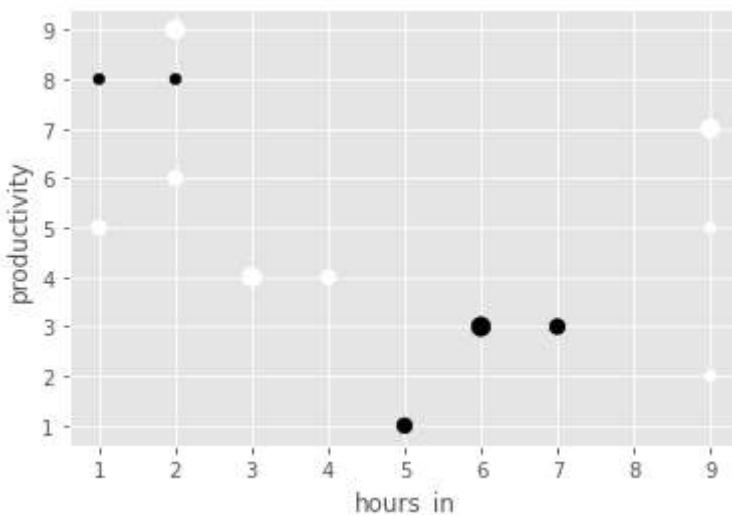
In [82]:

```
df = pd.DataFrame({"productivity": [5, 2, 3, 1, 4, 5, 6, 7, 8, 3, 4, 8, 9],
                   "hours_in"      : [1, 9, 6, 5, 3, 9, 2, 9, 1, 7, 4, 2, 2],
                   "happiness"    : [2, 1, 3, 2, 3, 1, 2, 3, 1, 2, 2, 1, 3],
                   "caffienated"  : [0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0]})
```

```
df.plot.scatter("hours_in", "productivity", s = df.happiness * 30, c = df.caffienated)
```

Out[82]:

```
<AxesSubplot:xlabel='hours_in', ylabel='productivity'>
```



**58.** What if we want to plot multiple things? Pandas allows you to pass in a matplotlib *Axis* object for plots, and plots will also return an *Axis* object.

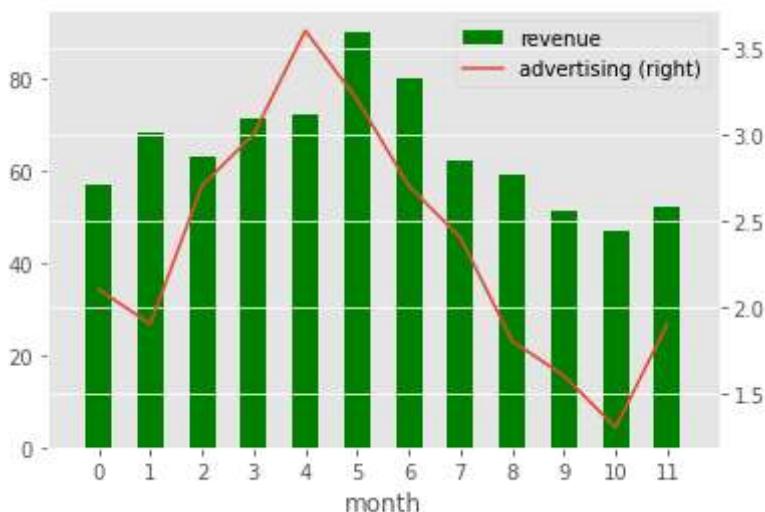
Make a bar plot of monthly revenue with a line plot of monthly advertising spending (numbers in millions)

```
df = pd.DataFrame({"revenue": [57, 68, 63, 71, 72, 90, 80, 62, 59, 51, 47, 52],
                   "advertising":
                   [2.1, 1.9, 2.7, 3.0, 3.6, 3.2, 2.7, 2.4, 1.8, 1.6, 1.3, 1.9],
                   "month": range(12)
                  })
```

```
In [83]: df = pd.DataFrame([{"revenue": [57, 68, 63, 71, 72, 90, 80, 62, 59, 51, 47, 52],
                        "advertising": [2.1, 1.9, 2.7, 3.0, 3.6, 3.2, 2.7, 2.4, 1.8, 1.6, 1.3, 1.9],
                        "month": range(12)
                      })

ax = df.plot.bar("month", "revenue", color = "green")
df.plot.line("month", "advertising", secondary_y = True, ax = ax)
ax.set_xlim((-1,12))
```

Out[83]: (-1.0, 12.0)



Now we're finally ready to create a candlestick chart, which is a very common tool used to analyze stock price data. A candlestick chart shows the opening, closing, highest, and lowest price for a stock during a time window. The color of the "candle" (the thick part of the bar) is green if the stock closed above its opening price, or red if below.



This was initially designed to be a pandas plotting challenge, but it just so happens that this type of plot is just not feasible using pandas' methods. If you are unfamiliar with matplotlib, we have provided a function that will plot the chart for you so long as you can use pandas to get the data into the correct format.

Your first step should be to get the data in the correct format using pandas' time-series grouping function. We would like each candle to represent an hour's worth of data. You can write your own aggregation function which returns the open/high/low/close, but pandas has a built-in which also does this.

The below cell contains helper functions. Call `day_stock_data()` to generate a DataFrame containing the prices a hypothetical stock sold for, and the time the sale occurred. Call `plot_candlestick(df)` on your properly aggregated and formatted stock data to print the candlestick chart.

In [84]:

```

import numpy as np
def float_to_time(x):
    return str(int(x)) + ":" + str(int(x%1 * 60)).zfill(2) + ":" + str(int(x*60 % 1 * 6))

def day_stock_data():
    #NYSE is open from 9:30 to 4:00
    time = 9.5
    price = 100
    results = [(float_to_time(time), price)]
    while time < 16:
        elapsed = np.random.exponential(.001)
        time += elapsed
        if time > 16:
            break
        price_diff = np.random.uniform(.999, 1.001)
        price *= price_diff
        results.append((float_to_time(time), price))

    df = pd.DataFrame(results, columns = ['time','price'])
    df.time = pd.to_datetime(df.time)
    return df

#Don't read me unless you get stuck!
def plot_candlestick(agg):
    """
    agg is a DataFrame which has a DatetimeIndex and five columns: ["open","high","low"]
    """
    fig, ax = plt.subplots()
    for time in agg.index:
        ax.plot([time.hour] * 2, agg.loc[time, ["high","low"]].values, color = "black")
        ax.plot([time.hour] * 2, agg.loc[time, ["open","close"]].values, color = agg.lo
  
```

```
ax.set_xlim((8,16))
ax.set_ylabel("Price")
ax.set_xlabel("Hour")
ax.set_title("OHLC of Stock Value During Trading Day")
plt.show()
```

- 59.** Generate a day's worth of random stock data, and aggregate / reformat it so that it has hourly summaries of the opening, highest, lowest, and closing prices

In [86]:

```
df = day_stock_data()
df.head()

df.set_index("time", inplace = True)
agg = df.resample("H").ohlc()
agg.columns = agg.columns.droplevel()
agg[ "color" ] = (agg.close > agg.open).map({True:"green",False:"red"})
agg.head()
```

Out[86]:

	open	high	low	close	color
time					
2021-12-19 09:00:00	100.000000	101.106171	99.815487	100.424561	green
2021-12-19 10:00:00	100.518640	102.800034	100.518640	101.383433	green
2021-12-19 11:00:00	101.440564	102.487595	99.875375	101.368365	red
2021-12-19 12:00:00	101.346975	101.751106	96.652501	96.957487	red
2021-12-19 13:00:00	96.872126	97.235394	94.509655	95.197672	red

- 60.** Now that you have your properly-formatted data, try to plot it yourself as a candlestick chart. Use the `plot_candlestick(df)` function above, or matplotlib's [plot documentation](#) if you get stuck.

In [87]:

```
plot_candlestick(agg)
```

