Python Programming for Data Science

Week 41, Friday

External modules: Pandas

Pandas: Series

pandas

What is pandas

pandas

What is pandas

- A library for data manipulation and analysis.

pandas

What is pandas

- A library for data manipulation and analysis.
- Make it easy to work with structured tables of numbers

pandas

What is pandas

- A library for data manipulation and analysis.
- Make it easy to work with structured tables of numbers
- Much faster than writing loops in Python

pandas

What is pandas

- A library for data manipulation and analysis.
- Make it easy to work with structured tables of numbers
- Much faster than writing loops in Python

In short: whenever you have a list of numbers, consider using numpy and pandas.

pandas

What is pandas

- A library for data manipulation and analysis.
- Make it easy to work with structured tables of numbers
- Much faster than writing loops in Python

In short: whenever you have a list of numbers, consider using numpy and pandas.

Main difference from numpy: data is labeled.

pandas - importing

```
import pandas
```

pandas - importing

```
import pandas
```

or:

```
import pandas as pd
```

pandas: two data types

Pandas is centered around two data types

- Series
    - Like a 1D numpy array - but with labels
- Dataframe
    - A 2D data structure with labeled columns

pandas - Series

A `Series` is basically a numpy array with an index which defines labels for each entry

```
np_array = np.array([1.0, 2.0, 3.0])

# convert numpy array to pandas series
pd_series = pd.Series(np_array, index=['a', 'b', 'c'])

print(pd_series)
```

output

```
a     1.0
b     2.0
c     3.0
dtype: float64
```

pandas - Series - from a dict

You can also create a Series from a dict

```
# Convert dictionary to pandas series
pd_series = pd.Series({'a':1.0, 'b':2.0, 'c':3.0})
print(pd_series)
```

output

```
a    1.0
b    2.0
c    3.0
dtype: float64
```

pandas - Series - behave as arrays and dicts

Series behave similar to numpy arrays and dicts

```
# Just like numpy arrays, I can operate on all elements at once
# ... but note how the labels stay aligned
print(pd_series * 2)
```

output

```
a     2.0
b     4.0
c     6.0
dtype: float64
```

pandas - Series - behave as arrays and dicts

`Series` behave similar to numpy arrays and dicts

```
# Just like numpy arrays, I can operate on all elements at once
# ... but note how the labels stay aligned
print(pd_series * 2)
```

output

```
a     2.0
b     4.0
c     6.0
dtype: float64
```

```
print(np.log(pd_series))    # I can also use functions from numpy
```

output

```
a     0.000000
b     0.693147
c     1.098612
dtype: float64
```

pandas - Series - behave as arrays and dicts

`Series` behave similar to numpy arrays and dicts

```
# Just like numpy arrays, I can operate on all elements at once
# ... but note how the labels stay aligned
print(pd_series * 2)
```

output

```
a     2.0
b     4.0
c     6.0
dtype: float64
```

```
print(np.log(pd_series))    # I can also use functions from numpy
```

output

```
a     0.000000
b     0.693147
c     1.098612
dtype: float64
```

```
# Just like dicts, I can ask for a specific key
print(pd_series['b'])
```

output

```
2.0
```

pandas - Series - difference to numpy

Labels are used for alignment

```
print(pd_series + pd_series[1:])
```

output

```
a     NaN
b     4.0
c     6.0
dtype: float64
```

Very powerful! - no worrying about adding pears to apples

pandas - Series - difference to numpy

Labels are used for alignment

```
print(pd_series + pd_series[1:])
```

output

```
a     NaN
b     4.0
c     6.0
dtype: float64
```

Very powerful! - no worrying about adding pears to apples

Note that all indices are retained - unless you explicitly drop them:

```
print((pd_series + pd_series[1:]).dropna())
```

output

```
b     4.0
c     6.0
dtype: float64
```

pandas - accessors

Pandas provides specific support for Series of certain types, through so-called Accessors.

Data type	Accessor
String	str
Categorical	cat
Datetime	dt
Sparse	sparse

For strings, this makes the usual methods available:

```
series = pd.Series({'a':'I', 'b':'love', 'c':'python'})
print(pd_series.str.upper())
```

output

```
a          I
b       LOVE
c     PYTHON
dtype: object
```

pandas - accessors

Pandas provides specific support for Series of certain types, through so-called Accessors.

Data type	Accessor
String	str
Categorical	cat
Datetime	dt
Sparse	sparse

For strings, this makes the usual methods available:

```
series = pd.Series({'a':'I', 'b':'love', 'c':'python'})
print(pd_series.str.upper())
```

output

```
a          I
b       LOVE
c     PYTHON
dtype: object
```

You can explicitly convert a series into strings using `.astype('string')`

pandas - Categorical

In addition to the types known from numpy, pandas also has a categorical type

Can be created in different ways:

```
series = pd.Series(['male', 'female', 'female'], dtype="category")
series = pd.Series(['male', 'female', 'female']).astype('category')
series = pd.Series(pd.Categorical(['male', 'female', 'female']))
series = pd.Series(pd.Categorical(['male', 'female', 'female'],
                                  categories=['female', 'male']))
```

Category specific functionality are accessible through `.cat`:

```
print(series.cat.categories)
```

output

```
Index(['female', 'male'], dtype='object')
```

pandas - Series - Exercise

1. Use range to create a list with values from 0 to 99 and use it to initialize a Series
2. Convert the series to type string, and calculate the length of the entries

pandas - Series - Exercise - solution

1. Use range to create a list with values from 0 to 99 and use it to initialize a Series

```
# Create range series
series = pd.Series(range(100))
```

2. Convert the series to type string, and calculate the length of the entries

```
# Convert to string and calculate lengths
series.astype('string').str.len()
```

Pandas: Dataframes

pandas - Dataframe

- The main data type in pandas
- Equivalent to a SAS dataset, an R data frame or an SQL table
- Can be thought of as a `dict` of `Series`

pandas - Dataframe - from numpy array

A dataframe has to sets of labels

index: row labels

columns: column labels

Dataframes can be created from numpy arrays by providing these sets of labels

```
np_array = np.arange(6).reshape((2,3))
df = pd.DataFrame(np_array, index=['a', 'b'], columns=['col1', 'col2', 'col3'])
print(df)
```

output

```
   col1  col2  col3
a     0     1     2
b     3     4     5
```

pandas - Dataframe - from dictionary of lists

You can also initialize from a dictionary of lists

```
dict_of_lists = {'col1': [0,3], 'col2': [1,4], 'col3': [2,5]}
df = pd.DataFrame(dict_of_lists, index=['a', 'b'])
print(df)
```

output

```
   col1  col2  col3
a     0     1     2
b     3     4     5
```

pandas - Dataframe - from dictionary of lists

You can also initialize from a dictionary of lists

```
dict_of_lists = {'col1': [0,3], 'col2': [1,4], 'col3': [2,5]}
df = pd.DataFrame(dict_of_lists, index=['a', 'b'])
print(df)
```

output

```
   col1  col2  col3
a     0     1     2
b     3     4     5
```

If `index` is not specified, it will use [0,1, ...]

```
df = pd.DataFrame(dict_of_lists)
print(df)
```

output

```
   col1  col2  col3
0     0     1     2
1     3     4     5
```

pandas - Dataframe - from list of dictionaries

You can also initialize from a lists of dictionaries

```
list_of_dicts = [{'col1': 0, 'col2': 1, 'col3': 2},
                 {'col1': 3, 'col2': 4, 'col3': 5}]
df = pd.DataFrame(list_of_dicts, index=['a', 'b'])
print(df)
```

output

```
   col1  col2  col3
a     0     1     2
b     3     4     5
```

pandas - Dataframe - from list of dictionaries

You can also initialize from a lists of dictionaries

```
list_of_dicts = [{'col1': 0, 'col2': 1, 'col3': 2},
                 {'col1': 3, 'col2': 4, 'col3': 5}]
df = pd.DataFrame(list_of_dicts, index=['a', 'b'])
print(df)
```

```
   col1  col2  col3                                                  output
a     0     1     2
b     3     4     5
```

There are many more ways to initialize DataFrames...

pandas - Dataframe: index, columns, values

The labels and values can be accessed using `.index`, `.columns`, and `.values`

```
np_array = np.arange(6).reshape((3,2))
df = pd.DataFrame(np_array,
                  index=['a', 'b', 'c'],
                  columns=['col1', 'col2'])
print(df.index)
print(df.columns)
print(df.values)
```

output

```
Index(['a', 'b', 'c'], dtype='ob
Index(['col1', 'col2'], dtype='c
[[0 1]
 [2 3]
 [4 5]]
```

pandas - Indexing into a dataframe

General - also supports slicing, etc (see later slide):

Get by label:	`df.loc[row_label, col_label]`
Get by index:	`df.iloc[row_index, col_index]`

pandas - Indexing into a dataframe

General - also supports slicing, etc (see later slide):

Get by label:	`df.loc[row_label, col_label]`
Get by index:	`df.iloc[row_index, col_index]`

Faster - for lookup of single values:

Get by label:	`df.at[row_label, col_label]`
Get by index:	`df.iat[row_index, col_index]`

pandas - Other ways to index into a dataframe

Indexing directly into a dataframe:

Get column by label:	`df[col_label]`	→Series
Slice rows by index range:	`df[start:end]`	→DataFrame
Filter by boolean list/array	`df[bool_list]`	→DataFrame

This can be a bit confusing - better to stick with `loc`, `iloc`, `at`, `iat`

pandas - `.loc`: selecting by labels

The `.loc` attribute is the primary access method

- It selects **by label!**
- Ranges are allowed: all labels between **and including** both endpoints are included

```python
np_array = np.arange(6).reshape((3,2))
df = pd.DataFrame(np_array, index=['a', 'b', 'c'],
                  columns=['col1', 'col2'])
print(df)



# note that both endpoints are included
print(df.loc['a':'b'])
```

output

```
   col1  col2
a     0     1
b     2     3
c     4     5

   col1  col2
a     0     1
b     2     3
```

pandas - `.loc`: selecting by labels (2)

You can select along both rows and columns with `.loc`

```
np_array = np.arange(6).reshape((3,2))
df = pd.DataFrame(np_array, index=['a', 'b', 'c'],
                  columns=['col1', 'col2'])
print(df)




print(df.loc['a':'b', :'col1'])
```

```
output


   col1  col2
a     0     1
b     2     3
c     4     5

   col1
a     0
b     2
```

pandas - `.loc`: selecting by labels (2)

You can select along both rows and columns with `.loc`

```
np_array = np.arange(6).reshape((3,2))
df = pd.DataFrame(np_array, index=['a', 'b', 'c'],
                  columns=['col1', 'col2'])
print(df)




print(df.loc['a':'b', :'col1'])
```

```
output



   col1  col2
a     0     1
b     2     3
c     4     5

   col1
a     0
b     2
```

Q: What would happen if I removed the ':' before 'col1'?

pandas - `.loc`: selecting by labels (2)

You can select along both rows and columns with `.loc`

```python
np_array = np.arange(6).reshape((3,2))
df = pd.DataFrame(np_array, index=['a', 'b', 'c'],
                  columns=['col1', 'col2'])
print(df)




print(df.loc['a':'b', :'col1'])
```

```
output


   col1  col2
a     0     1
b     2     3
c     4     5

   col1
a     0
b     2
```

Q: What would happen if I removed the ':' before 'col1'?

A: It returns a series

pandas - `.loc`: selecting with a boolean array

```
np_array = np.arange(6).reshape((3,2))
df = pd.DataFrame(np_array, index=['a', 'b', 'c'],
                  columns=['col1', 'col2'])
print(df)




print(df.loc[:, 'col1']>2)



df.loc[df.loc[:, 'col1']>2] = 0
print(df)
```

```
                                     output


   col1  col2
a     0     1
b     2     3
c     4     5

a    False
b    False
c     True
Name: col1, dtype:

   col1  col2
a     0     1
b     2     3
c     0     0
```

pandas - reading/writing in various formats

`read_table()`	
`read_csv()`	`to_csv()`
`read_html()`	`to_html()`
`read_json()`	`to_json()`
`read_hdf()`	`to_hdf()`
`read_excel()`	`to_excel()`
`read_sas()`	
`read_sql()`	`to_sql()`

...

pandas - Exercise 2

1. Download: https://wouterboomsma.github.io/ppds2021/data/british-english
2. Read it into a DataFrame (please use the `keep_default_na=False` option to `read_table`)
3. Figure out how to assign a different column name - e.g. 'words'
4. Figure out how to select all rows starting with the letter A

pandas - Exercise 2 - solution

2. Read it into a DataFrame

```
df = pd.read_table('british-english', keep_default_na=False, header=None)
```

3. Figure out how to assign a different column name

```
df.columns = ['words']
```

4. Figure out how to select all rows starting with the letter A

```
df.loc[df['words'].str.startswith('A')]
```

pandas - DataFrames - Adding columns

Add a new column by assigning to the relevant label

```
np_array = np.arange(6).reshape((3,2))
df = pd.DataFrame(np_array, index=['a', 'b', 'c'], columns=['col1', 'col2'])
df.loc[:,'col3'] = df.loc[:,'col1']   # Adding 'col3' - copy of `col1`
```

...or use the `.assign()` method

```
df.assign(col4 = df.loc[:,'col1'])
```

...or `.insert()` (inserts at a specific location)

```
df.insert(0, 'col0', df.loc[:,'col1'])
```

pandas - DataFrames - Adding columns (2)

If columns don't have the expected index, missing elements will be set to NaN

```
np_array = np.arange(6).reshape((3,2))
df = pd.DataFrame(np_array, index=['a', 'b', 'c'], columns=['col1', 'col2'])
print(df)
df.loc[:,'col3'] = df.loc['b':,'col1']    # skipping first row
print(df)
```

output

```
   col1  col2
a     0     1
b     2     3
c     4     5

   col1  col2  col3
a     0     1   NaN
b     2     3   2.0
c     4     5   4.0
```

pandas - DataFrames - Adding rows

Add a new row by assigning to the relevant label

```
np_array = np.arange(6).reshape((3,2))
df = pd.DataFrame(np_array, index=['a', 'b', 'c'], columns=['col1', 'col2'])
df.loc['d',:] = df.loc['a',:]   # Adding 'd' - copy of `a`
```

...or by using the `.append()` method

```
np_array = np.arange(6).reshape((3,2))
df = pd.DataFrame(np_array, index=['a', 'b', 'c'], columns=['col1', 'col2'])
df.append(df.loc['a',:])   # Adding copy of `a` - under the same name!
```

pandas - DataFrames - Adding rows

Add a new row by assigning to the relevant label

```
np_array = np.arange(6).reshape((3,2))
df = pd.DataFrame(np_array, index=['a', 'b', 'c'], columns=['col1', 'col2'])
df.loc['d',:] = df.loc['a',:]   # Adding 'd' - copy of `a`
```

...or by using the `.append()` method

```
np_array = np.arange(6).reshape((3,2))
df = pd.DataFrame(np_array, index=['a', 'b', 'c'], columns=['col1', 'col2'])
df.append(df.loc['a',:])   # Adding copy of `a` - under the same name!
```

Duplicate index values are allowed, but makes lookups slower.

pandas - DataFrames - data alignment

When operating with two dataframes, pandas aligns on both row and column labels

```
np_array = np.arange(9).reshape((3,3))
df1 = pd.DataFrame(np_array)
df2 = pd.DataFrame(np_array[:2,:2])
print(df1)
print(df2)
print(df1+df2)
```

```
output

   0  1  2
0  0  1  2
1  3  4  5
2  6  7  8

   0  1
0  0  1
1  3  4

     0    1    2
0  0.0  2.0  NaN
1  6.0  8.0  NaN
2  NaN  NaN  NaN
```

pandas - DataFrames - descriptive statistics

Long list of standard operations: `.mean()`, `.std()`, `.var()`, `.min()`, `.max()`, `.sumsum()`, `.sumprod()`, ...

These functions generally take an `axis` argument, and a `skipna` argument (which default to True)

Missing values

As we've seen, pandas will insert NaN for missing values

Q: How are these dealt with during calculations?

Missing values

As we've seen, pandas will insert NaN for missing values

Q: How are these dealt with during calculations?

A: Unlike in numpy, pandas takes missing values into account

```
np_array = np.arange(9).reshape((3,3))
df1 = pd.DataFrame(np_array)
df2 = pd.DataFrame(np_array[:2,:2])
print(df1+df2)




print((df1+df2).sum(axis=0))
```

```
output



     0    1    2
0  0.0  2.0  NaN
1  6.0  8.0  NaN
2  NaN  NaN  NaN

0     6.0
1    10.0
2     0.0
dtype: float64
```

Missing values

As we've seen, pandas will insert NaN for missing values

Q: How are these dealt with during calculations?

A: Unlike in numpy, pandas takes missing values into account

```
np_array = np.arange(9).reshape((3,3))
df1 = pd.DataFrame(np_array)
df2 = pd.DataFrame(np_array[:2,:2])
print(df1+df2)




print((df1+df2).sum(axis=0))
```

output

```
     0    1    2
0  0.0  2.0  NaN
1  6.0  8.0  NaN
2  NaN  NaN  NaN

0     6.0
1    10.0
2     0.0
dtype: float64
```

You can also specify the minimum number of non-NaN values, using the `min_count` option.

Missing values (2)

You can detect missing values using `.isna()` and `.notna()`

```
np_array = np.arange(9).reshape((3,3))
df1 = pd.DataFrame(np_array)
df2 = pd.DataFrame(np_array[:2,:2])
print(df1+df2)




print((df1+df2).isna())
```

output

```
     0    1    2
0  0.0  2.0  NaN
1  6.0  8.0  NaN
2  NaN  NaN  NaN

       0      1      2
0  False  False   True
1  False  False   True
2   True   True   True
```

Missing values (3)

You can replace missing values using `.fillna()`

```
np_array = np.arange(9).reshape((3,3))
df1 = pd.DataFrame(np_array)
df2 = pd.DataFrame(np_array[:2,:2])
print(df1+df2)




print((df1+df2).fillna(0.0))
```

```
output



     0    1   2
0  0.0  2.0 NaN
1  6.0  8.0 NaN
2  NaN  NaN NaN

     0    1    2
0  0.0  2.0  0.0
1  6.0  8.0  0.0
2  0.0  0.0  0.0
```

Missing values (4)

You can also:

- Drop missing values using `.dropna()`
- Replace missing by interpolation `.interpolate()`
- ...

pandas - sorting labels

`.sort_index()` sorts either by row or column labels:

```
np_array = np.arange(6).reshape((3,2))
df = pd.DataFrame(np_array,
                  index=['a', 'b', 'c'],
                  columns=['col1', 'col2'])
print(df.sort_index(axis=1, ascending=False))
```

output

```
   col2  col1
a     1     0
b     3     2
c     5     4
```

pandas - sorting labels

`.sort_index()` sorts either by row or column labels:

```
np_array = np.arange(6).reshape((3,2))
df = pd.DataFrame(np_array,
                  index=['a', 'b', 'c'],
                  columns=['col1', 'col2'])
print(df.sort_index(axis=1, ascending=False))
```

output

```
   col2  col1
a     1     0
b     3     2
c     5     4
```

There is an `inplace` option which changes the original instead of returning a new data frame

pandas - sorting values

`.sort_values(by=`*label*`)` sorts either by row or column labels:

```
np_array = np.arange(6).reshape((3,2))
df = pd.DataFrame(np_array,
                  index=['a', 'b', 'c'],
                  columns=['col1', 'col2'])
print(df.sort_values(by='col2', ascending=False))
```

output

```
   col1  col2
c     4     5
b     2     3
a     0     1
```

pandas - sorting values

`.sort_values(by=`*label*`)` sorts either by row or column labels:

```
np_array = np.arange(6).reshape((3,2))
df = pd.DataFrame(np_array,
                  index=['a', 'b', 'c'],
                  columns=['col1', 'col2'])
print(df.sort_values(by='col2', ascending=False))
```

```
output

   col1  col2
c     4     5
b     2     3
a     0     1
```

There is an `inplace` option which changes the original instead of returning a new data frame

pandas - sorting values

`.sort_values(by=`*label*`)` sorts either by row or column labels:

```
np_array = np.arange(6).reshape((3,2))
df = pd.DataFrame(np_array,
                  index=['a', 'b', 'c'],
                  columns=['col1', 'col2'])
print(df.sort_values(by='col2', ascending=False))
```

```
output

   col1  col2
c     4     5
b     2     3
a     0     1
```

There is an `inplace` option which changes the original instead of returning a new data frame

You can sort by multiple labels by providing a list to by=

pandas - DataFrames - concatenating

```
np_array = np.arange(6).reshape((3,2))
df1 = pd.DataFrame(np_array, index=['a', 'b', 'c'],
                   columns=['col1', 'col2'])
print(df1)

df2 = pd.DataFrame(np_array[:2, :1],
                   index=['a', 'b'],
                   columns=['col3', ])
print(df2)



print(pd.concat([df1, df2], axis=1))
```

output

```
   col1  col2
a     0     1
b     2     3
c     4     5

   col3
a     0
b     2

   col1  col2  col3
a     0     1   0.0
b     2     3   2.0
c     4     5   NaN
```

pandas - DataFrames - merging

`.merge()` is an efficient implementation of SQL-like merge operations

```
df1 = pd.DataFrame({'id' : [1, 2, 3],
                    'gender' : pd.Categorical(['male', 'female', 'female'])})
df2 = pd.DataFrame({'id' : [1, 2, 3],
                    'name' : pd.Categorical(['Bob', 'Alice', 'Anna'])})
pd.merge(df1, df2, on='id')
```

output

```
   gender  id   name
0    male   1    Bob
1  female   2  Alice
2  female   3   Anna
```

You can also merge on the index labels

```
df1 = pd.DataFrame({'gender' : pd.Categorical(['male', 'female', 'female'])})
df2 = pd.DataFrame({'name' : pd.Categorical(['Bob', 'Alice', 'Anna'])})
pd.merge(df1, df2, left_index=True, right_index=True)
```

pandas - DataFrames - grouping

The `.groupby()` groups by value

```
df = pd.DataFrame({'name' : pd.Categorical(['Bob', 'Alice', 'Anna']),
                   'gender' : pd.Categorical(['male', 'female', 'female']),
                   'height': [170, 180, 165]})
gender_grps = df.groupby('gender')
print(gender_grps)
```

output

```
<pandas.core.groupby.DataFrameGroupBy object at 0x11861bd30>
```

You can also group by a dynamically created series:

```
df.groupby(df['name'].str.len())
```

pandas - DataFrames - grouping

The `.groupby()` groups by value

```
df = pd.DataFrame({'name' : pd.Categorical(['Bob', 'Alice', 'Anna']),
                   'gender' : pd.Categorical(['male', 'female', 'female']),
                   'height': [170, 180, 165]})
gender_grps = df.groupby('gender')
print(gender_grps)
```

output

```
<pandas.core.groupby.DataFrameGroupBy object at 0x11861bd30>
```

You can also group by a dynamically created series:

```
df.groupby(df['name'].str.len())
```

What can you do with these groups?

pandas - DataFrames - grouping (2)

1. The `.groups` attribute provides some information

```
print(gender_grps.groups)
```

output

```
{'female': Int64Index([1, 2], dtype='int64'), 'male': Int64Index([0], dtype='int64
```

2. You an also iterate over the groups

```
for name,grp in gender_grps:
    print(name)
    print(grp)
```

output

```
female
   gender  height   name
1  female     180  Alice
2  female     165   Anna
male
  gender  height name
0   male     170  Bob
```

pandas - DataFrames - grouping (3)

...or you can calculate summary statistics

```
print(gender_grps.agg({'height':np.average}))
```

output

```
        height
gender
female   172.5
male     170.0
```

pandas - grouping - Exercise

1. Download: https://wouterboomsma.github.io/ppds2021/data/british-english, and read it into a dataframe
2. Get pandas to group by first letter - and use this to count the words for each letter

pandas - grouping - Exercise - solution

1. Download: https://wouterboomsma.github.io/ppds2021/data/british-english, and read it into a dataframe

```
df = pd.read_table('british-english', keep_default_na=False, header=None)
```

2. Get pandas to group by first letter - and use this to count the words for each letter

```
df.groupby(df[0].str[0]).count()
```

pandas - working with time data

Pandas has been designed for use with time series data

You can create a time range using `pd.date_range()`, and use this as index in a `Series`

```
idx = pd.date_range('26/9/2018 08:00', periods=3, freq='H')
series = pd.Series(np.arange(len(idx)), index=idx)
print(series)
```

output

```
2018-09-26 08:00:00    0
2018-09-26 09:00:00    1
2018-09-26 10:00:00    2
Freq: H, dtype: int64
```

pandas - working with time data (2)

We can change the frequency using `.asfreq`

```
print(series.asfreq('30Min'))
```

output

```
2018-09-26 08:00:00    0.0
2018-09-26 08:30:00    NaN
2018-09-26 09:00:00    1.0
2018-09-26 09:30:00    NaN
2018-09-26 10:00:00    2.0
```

Use the `method` option to specify how to fill new values. Alternatively:

```
print(series.asfreq('30Min').interpolate())
```

output

```
2018-09-26 08:00:00    0.0
2018-09-26 08:30:00    0.5
2018-09-26 09:00:00    1.0
...
```

pandas - working with time data (2)

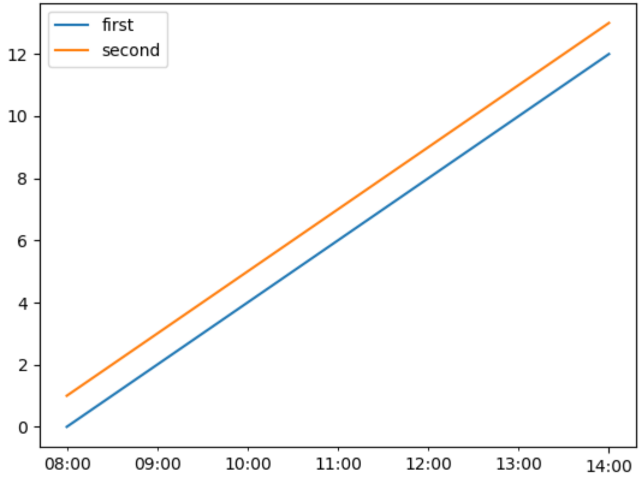
Too much to cover here, but you can:

- Using indexing and slicing on the time index
- Index with partial data strings
- Use BusinessDay as time unit - and define custom versions
- Automatically skip holidays
- Resampling time (time-based groupby)
- ...

pandas - plotting

Dataframes have associated plotting capabilities through matplotlib

```
idx = pd.date_range('26/9/2018 08:00',
                    periods=7, freq='H')
df = pd.DataFrame(np.arange(14).reshape([7,2]),
                  columns=['first', 'second'],
                  index=idx)
df.plot()
```

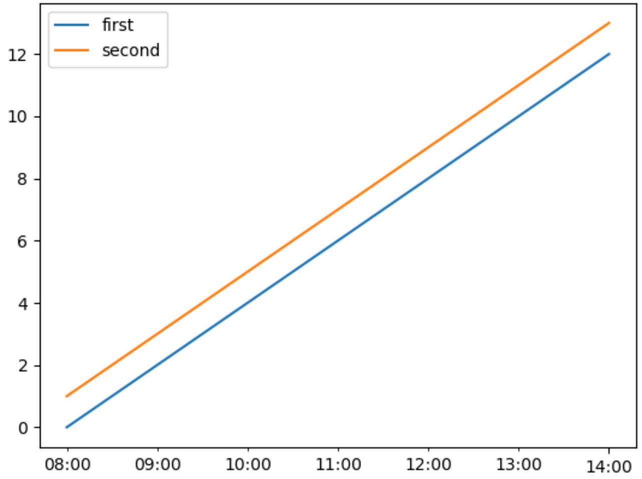


Note how the data frame labels are automatically used.

pandas - plotting

Dataframes have associated plotting capabilities through matplotlib

```
idx = pd.date_range('26/9/2018 08:00',
                    periods=7, freq='H')
df = pd.DataFrame(np.arange(14).reshape([7,2]),
                  columns=['first', 'second'],
                  index=idx)
df.plot()
```



Note how the data frame labels are automatically used.

Lots of other plot types available, using attributes under plot: `df.plot.bar`, `df.plot.scatter`, etc

pandas - plotting - exercise

1. Create a histogram of the word counts from the previous exercise.
2. Bonus: Try to get meaningful labels on the x-axis and in the legend.

pandas - plotting - exercise - solution

1. Create a histogram of the word counts from the previous exercise.

```
# From previous exercise
df = pd.read_table('british-english', keep_default_na=False, header=None)
result = df.groupby(df.loc[:,0].str[0]).count()

# This exercise:
result = result.rename_axis(index='letter')  # rename index axis
result.columns = ['word count']   # Give column a more informative name
result.plot.bar()
```