Data Analysis in Python

In this session we will learn how to properly utilize python's pandas (https://pandas.pydata.org/) library for data transforming, cleaning, filtering and exploratory data analysis.

Pandas

Python's Data Analysis Library

Python has long been great for data munging and preparation, but less so for data analysis and modeling. *Pandas* helps fill this gap, enabling you to carry out your entire data analysis workflow in Python.

Pandas is built on top of *numpy* aiming at providing higher-level functionality as well as a new data structure that works well with tabular data with heterogenous-typed columns (e.g. Excel spreadsheets, SQL tables).

Data Structures

Pandas introduces two new data structures to Python: the **Series** and the **DataFrame**. Both of which are built on top of NumPy.

Series

A **series**, in *pandas* is a one-dimensional *ndarray* with axis labels. The axis labels are collectively referred to as the **index**. The labels facilitate in allowing us to refer to the elements in the series either by their position (like in a list or an array) or by their label (like in a dictionary).

The basic method to create a `pd.Series` is to call:

```
s = pd.Series(data, index=index)
```

where *data* is most commonly a dictionary (where the keys will be used as the `index` and the values as the elements) or a `numpy.array` and `index` is a *list* of labels.

In [1]:
```
from __future__ import print_function
import pandas as pd  # for simplicity we usually refer to pandas as pd
import numpy as np

s = pd.Series([1,3,5,np.nan,6,8], index=['a', 'b', 'c', 'd', 'e', 'f'])
# By passing a list as the only argument in series, we let pandas create
print(s)
```

```
a    1.0
b    3.0
c    5.0
d    NaN
e    6.0
f    8.0
dtype: float64
```

Like arrays, a series can only have one `dtype` (in this case `float64`).

As we mentioned previously, indexing elements in the *Series* can be done either through their position or through their label.

In [2]:
```
print(s[4])     # position
print(s['e'])   # label
```

```
6.0
6.0
```

If we don't set an `index` during the creation of the *Series*, the labels will be set to the position of each element.

In [3]:
```
s = pd.Series([1,3,5,np.nan,6,8])
print(s)
```

```
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
dtype: float64
```

The last is the most common use of a series.

We can easily keep the underlying `np.array` containing just the values of the *Series*.

In [4]:
```
s.values  # a np.array with the values of the Series
```

Out[4]:
```
array([ 1.,  3.,  5., nan,  6.,  8.])
```

A **DataFrame** is a two-dimensional labeled data structure with columns of potentially different types. You can think of it like a spreadsheet. It is organized in such a way that it is essentially a collection of `pd.Series`, where each series is a column. This way each column must have a **single** data type, but the data type can **differ from column to column**.

A *DataFrame* can have labels for both its rows and its columns, however we usually prefer to label **only the columns** and leave the rows to have their position as their labels.

The easiest way to create a *DataFrame* is to pass in a dictionary of objects.

In [5]:
```
df = pd.DataFrame({'A' : 1,  # repeats integer for the length of the data
                   'B' : pd.Timestamp('20190330'),  # timestamp datatype,
                   'C' : pd.Series(range(4), dtype='float32'),  # creates
                   'D' : np.array([3] * 4,dtype='int32'),  # np.array as
                   'E' : pd.Categorical(["test","train","test","train"]),
                   'F' : 'foo' })  # string, repeats it for the length of

df  # renders better in jupyter if we don't use print
```

Out[5]:

	A	B	C	D	E	F
0	1	2019-03-30	0.0	3	test	foo
1	1	2019-03-30	1.0	3	train	foo
2	1	2019-03-30	2.0	3	test	foo
3	1	2019-03-30	3.0	3	train	foo

DataFrame inspection

In most cases the *DataFrames* are thousands of rows long, we can't view all the data at once.

- Look at the **first** entries.

In [6]:
```
df.head()  # prints first entries (by default 5)
```

Out[6]:

	A	B	C	D	E	F
0	1	2019-03-30	0.0	3	test	foo
1	1	2019-03-30	1.0	3	train	foo
2	1	2019-03-30	2.0	3	test	foo
3	1	2019-03-30	3.0	3	train	foo

- Look at the **last** entries.

In [7]:
```
df.tail(3)  # prints last 3 entries
```

Out[7]:

	A	B	C	D	E	F
1	1	2019-03-30	1.0	3	train	foo
2	1	2019-03-30	2.0	3	test	foo
3	1	2019-03-30	3.0	3	train	foo

- Look at entries at **random**.

In [8]:
```
df.sample(2)  # prints two random entries
```

Out[8]:

	A	B	C	D	E	F
3	1	2019-03-30	3.0	3	train	foo
2	1	2019-03-30	2.0	3	test	foo

Information about the *DataFrame*

The two main attributes of a *DataFrame* are:

- Its `shape`. *DataFrames* are always two-dimensional, so the only information this provides is the **number of rows and samples**.
- Its `dtypes`, which shows the data type of each of the columns.

In [9]:
```
print('shape:', df.shape)  # prints the shape of the dataframe
print(df.dtypes)           # prints the data type of each column
```

```
shape: (4, 6)
A              int64
B    datetime64[ns]
C           float32
D             int32
E          category
F            object
dtype: object
```

Another important attribute of the *DataFrame* is the labelling on its rows and columns.

In [10]:
```
print('Row names:   ', df.index)
print('Column names:', df.columns)
```

```
Row names:    RangeIndex(start=0, stop=4, step=1)
Column names: Index(['A', 'B', 'C', 'D', 'E', 'F'], dtype='object')
```

Statistical summary of numeric columns

We can also easily view a statistical description of our data (only the columns with numeric data types).

In [11]:
```
df.describe()  # only numerical features appear when doing this
```

Out[11]:

	A	C	D
count	4.0	4.000000	4.0
mean	1.0	1.500000	3.0
std	0.0	1.290994	0.0
min	1.0	0.000000	3.0
25%	1.0	0.750000	3.0
50%	1.0	1.500000	3.0
75%	1.0	2.250000	3.0
max	1.0	3.000000	3.0

Indexing data

Since *DataFrames* support both indexing through labels and through position we have two main ways of getting an item.

Positional indexing.

This is done through `.iloc`, which requires two arguments: the position of the desired element's row and the position of its column. `.iloc` essentially allows us to use the *DataFrame* as an array.

In [12]:
```
df.iloc[3, 2]  # element in the 4th row of the 3rd column
```

Out[12]:
```
3.0
```

Slicing works the same way it does in *numpy*.

In [13]:
```
df.iloc[::2, -3:]  # odd rows, last three columns
```

Out[13]:

	D	E	F
0	3	test	foo
2	3	test	foo

As does indexing through lists.

In [14]:
```
df.iloc[[0, 3], [1, 3, 4]]  # 1st and 4th row; 2nd, 4th and 5th columns
```

Out[14]:

	B	D	E
0	2019-03-30	3	test
3	2019-03-30	3	train

Indexing with labels

We can use the row and column labels to access an element through `.loc`. Remember, if we haven't assigned any labels to the rows, their labels will be the same as their position.

In [15]:
```
df.loc[3, 'C']  # element in the row with the label 3 and the column with
```

Out[15]:
```
3.0
```

Slicing also works!

In [16]:
```
df.loc[::2, 'B':'D']  # odd rows, columns 'B' through 'D'
```

Out[16]:

	B	C	D
0	2019-03-30	0.0	3
2	2019-03-30	2.0	3

And even indexing through lists.

In [17]:
```
df.loc[[0, 3], ['B', 'D', 'E']]  # 1st and 4th row; columns 'B', 'D', and
```

Out[17]:

	B	D	E
0	2019-03-30	3	test
3	2019-03-30	3	train

Note that `.loc` **included** `'D'` in its slice!

Without locators

Columns

Pandas offers an easier way of slicing one or more columns from a *DataFrame*.

In [18]:
```
df['B']  # get the column 'B'
```

Out[18]:
```
0   2019-03-30
1   2019-03-30
2   2019-03-30
3   2019-03-30
Name: B, dtype: datetime64[ns]
```

In [19]:
```
df[['B', 'D', 'E']]  # get a slice of the columns 'B', 'D' and 'E'
```

Out[19]:

	B	D	E
0	2019-03-30	3	test
1	2019-03-30	3	train
2	2019-03-30	3	test
3	2019-03-30	3	train

Note that if we slice a single column it will return a `pd.Series`, but if we slice more we'll get a `pd.DataFrame`.

If we wanted to get a `pd.DataFrame` with a single column we could use this syntax:

In [20]:
```
df[['B']]  # get a dataframe containing only the column 'B'
```

Out[20]:

	B
0	2019-03-30
1	2019-03-30
2	2019-03-30
3	2019-03-30

Pandas also allows us to slice columns with this syntax:

```
df.B  # gets the column 'B'

# Equivalient to:
df['B']
```

However, it is **not** recommended!

Slicing rows

We can easily slice rows like this:

```
df[:2]   # first two rows
df[-3:]  # last three rows
df[1:2]  # second row
```

However, if we try index a single row, it will raise an error (because it will be looking for a column named 2).

```
df[2]  # KeyError

# Instead use
df.loc[2]
# or
df.iloc[2]
```

Filtering

Pandas' allows us to easily apply filters on the *DataFrame* with the same syntax we saw in the previous tutorial. Here it is a bit more intuitive, due to the naming scheme!

Single condition

Like in *numpy*, operations here (even logical) are performed element-wise and, if necessary, with broadcasting.

In [21]:
```
df['E'] == 'test'
```

Out[21]:
```
0     True
1    False
2     True
3    False
Name: E, dtype: bool
```

If we use the result of the logical condition above as an index, pandas will filter the rows based on the `True` or `False` value.

In [22]:
```
df[df['E'] == 'test']  # keeps the rows that have a value equal to 'test'
```

Out[22]:

	A	B	C	D	E	F
0	1	2019-03-30	0.0	3	test	foo
2	1	2019-03-30	2.0	3	test	foo

This leads to very a intuitive and syntactically simple application of filters.

Combining multiple conditions

To combine the outcome of more than one logical conditions we have to use the following symbols:

```
(cond1) & (cond2)   # logical AND
(cond1) | (cond2)   # logical OR
~ (cond1)           # logical NOT
```

Don't forget the parentheses!

In [23]:
```
df[(df['C'] > 1) | (df['E'] == 'test')]   # keeps the rows that have a val
                                          # in column 'E' or a value large
```

Out[23]:

	A	B	C	D	E	F
0	1	2019-03-30	0.0	3	test	foo
2	1	2019-03-30	2.0	3	test	foo
3	1	2019-03-30	3.0	3	train	foo

Adding / Deleting

Rows

To add a new row, we can use `.append()` .

In [24]:
```
# Adds a fifth row to the DataFrame:
df.append({'A': 3,
           'B': pd.Timestamp('20190331'),
           'C': 4.0,
           'D': -3,
           'E': 'train',
           'F': 'bar'},
          ignore_index=True)
```

Out[24]:

	A	B	C	D	E	F
0	1	2019-03-30	0.0	3	test	foo
1	1	2019-03-30	1.0	3	train	foo
2	1	2019-03-30	2.0	3	test	foo
3	1	2019-03-30	3.0	3	train	foo
4	3	2019-03-31	4.0	-3	train	bar

Note that the length and the data types should be compatible! Because this syntax isn't very convenient we usually **avoid using it** altogether.

Keep in mind that this operation **isn't performed inplace**. Instead it returns a copy of the *DataFrame*! If we want to make the append permanent, we can always assign it to itself.

In [25]:
```
df = df.append({'A': 3,
                'B': pd.Timestamp('20190331'),
                'C': 4.0,
                'D': -3,
                'E': 'train',
                'F': 'bar'},
               ignore_index=True)

df
```

Out[25]:

	A	B	C	D	E	F
0	1	2019-03-30	0.0	3	test	foo
1	1	2019-03-30	1.0	3	train	foo
2	1	2019-03-30	2.0	3	test	foo
3	1	2019-03-30	3.0	3	train	foo
4	3	2019-03-31	4.0	-3	train	bar

Another option would be to add the row through `.loc`:

```
df.loc[len(df)] = [3, pd.Timestamp('20190331'), 4.0, -3, 'train',
'bar']
```

To delete a row from a *DataFrame* we can use `.drop()`:

```
row_label  # label of the row we want to delete

# Doesn't overwrite df, instead returns a copy:
df.drop(row_label)

# Overwrites df:
df = df.drop(row_label)
df.drop(row_label, inplace=True)
```

In [26]:
```
df = df.drop(2)  # drops the third row from the dataframe
```

Columns

We can add a new column in the *DataFrame* like we would an element in a dictionary. Just keep in mind that the dimensions must be compatible (e.g. we can't add 3 elements to a *DataFrame* with four rows).

In [27]:
```
df['G'] = [10, 22, -8, 13]
df
```

Out[27]:

	A	B	C	D	E	F	G
0	1	2019-03-30	0.0	3	test	foo	10
1	1	2019-03-30	1.0	3	train	foo	22
3	1	2019-03-30	3.0	3	train	foo	-8
4	3	2019-03-31	4.0	-3	train	bar	13

To delete a row we, again, can use `.drop(col_label, axis=1)`. The parameter `axis=1` tells pandas that we are looking to drop a column and that it should look for the key `col_name` in the columns.

In [28]:
```
df = df.drop('A', axis=1)  # drops column with the label 'A'
df
```

Out[28]:

	B	C	D	E	F	G
0	2019-03-30	0.0	3	test	foo	10
1	2019-03-30	1.0	3	train	foo	22
3	2019-03-30	3.0	3	train	foo	-8
4	2019-03-31	4.0	-3	train	bar	13

Sorting and rearranging

Transposing

This works exactly like in *numpy*.

In [29]:
```
df.T  # not inplace
```

Out[29]:

	0	1	3	4
B	2019-03-30 00:00:00	2019-03-30 00:00:00	2019-03-30 00:00:00	2019-03-31 00:00:00
C	0	1	3	4
D	3	3	3	-3
E	test	train	train	train
F	foo	foo	foo	bar
G	10	22	-8	13

Sorting

- By **value**

In [30]:
```
df = df.sort_values(by='G')  # sorts DataFrame according to values from c
df
```

Out[30]:

	B	C	D	E	F	G
3	2019-03-30	3.0	3	train	foo	-8
0	2019-03-30	0.0	3	test	foo	10
4	2019-03-31	4.0	-3	train	bar	13
1	2019-03-30	1.0	3	train	foo	22

Caution: that when performing operations that rearrange the rows, the row labels will **no longer match** the row positions!

To solve this issue, we can reset the labels to match the positions:

```
df.reindex()
```

This won't rearrange the *DataFrame* in any way; it will just **change the labelling of the rows**.

- By **index**

In [31]:
```
df = df.sort_index()
df
```

Out[31]:

	B	C	D	E	F	G
0	2019-03-30	0.0	3	test	foo	10
1	2019-03-30	1.0	3	train	foo	22
3	2019-03-30	3.0	3	train	foo	-8
4	2019-03-31	4.0	-3	train	bar	13

This **rearranged** the *DataFrame* so that the row labels are sorted!

By adding the argument `axis=1` we can perform these operations on the columns instead.

In [32]:
```
df.sort_index(axis=1, ascending=False)  # sort columns so that their name
```

Out[32]:

	G	F	E	D	C	B
0	10	foo	test	3	0.0	2019-03-30
1	22	foo	train	3	1.0	2019-03-30
3	-8	foo	train	3	3.0	2019-03-30
4	13	bar	train	-3	4.0	2019-03-31

Statistical information

These work only for numerical values. A sample of them are presented below, while there are [many more (https://pandas.pydata.org/pandas-docs/stable/api.html#api-dataframe-stats)](https://pandas.pydata.org/pandas-docs/stable/api.html#api-dataframe-stats) available.

In [33]:
```
print('Sum:')
print(df.sum())    # sum of each column
print('\nMean:')
print(df.mean())   # mean of each column
print('\nMin:')
print(df.min())    # minimum element of each column
print('\nMax:')
print(df.max())    # maximum element of each column
print('\nStandard deviation:')
print(df.std())    # standard deviation of each column
print('\nVariance:')
print(df.var())    # variance of each column
```

```
Sum:
C     8.0
D     6.0
G    37.0
dtype: float64

Mean:
C    2.00
D    1.50
G    9.25
dtype: float64

Min:
B    2019-03-30 00:00:00
C                      0
D                     -3
E                   test
F                    bar
G                     -8
dtype: object

Max:
B    2019-03-31 00:00:00
C                      4
D                      3
E                  train
F                    foo
G                     22
dtype: object

Standard deviation:
C     1.825742
D     3.000000
G    12.579746
dtype: float64

Variance:
C      3.333333
D      9.000000
G    158.250000
dtype: float64
```

Keep in mind that, contrary to *numpy*, *pandas* by default ignores `np.nan` values when performing operations.

Histograms

Another very convenient functionality offered by *pandas* is to find the unique values of a *Series* and count each value's number of occurrences.

```
Series.unique()        # returns an array of the unique values in
a pd.Series
Series.value_counts()  # returns the unique values along with the
ir number of occurrences
```

In [34]:
```
df['E'].unique()
```

Out[34]:
```
array(['test', 'train'], dtype=object)
```

In [35]:
```
df['E'].value_counts()
```

Out[35]:
```
train    3
test     1
Name: E, dtype: int64
```

Applying functions

One of the most powerful methods offered is `.apply()`. There are actually two different things that can be done by this method, depending on if it's called from a *DataFrame* or a *Series*.

DataFrame.apply()

When called from a *DataFrame*, `.apply()` applies a function to each of the *DataFrame's* columns **independently**. The built-in methods we saw previously produce similar results, the application of a function (e.g. `max`, `min`, `sum`) to every *DataFrame* column.

For example, how many **unique** values does each column have?

In [36]:
```
df['C'].unique()
```

Out[36]:
```
array([0., 1., 3., 4.])
```

The `len()` of this array shows *how many* unique values we have.

In [37]:
```
len(df['C'].unique())
```

Out[37]:
```
4
```

Now, can we apply this function to every column in the *DataFrame*?

In [38]:
```
# First, we need to write a function

def num_unique(series):
    # function that takes a series and returns the number of unique value
    return len(series.unique())

# Then apply in to each of the columns of the DataFrame
df.apply(num_unique)
```

Out[38]:
```
B    2
C    4
D    2
E    2
F    2
G    4
dtype: int64
```

It is common to write simple functions like these like **lambda functions** to save space.

In [39]:
```
df.apply(lambda s: len(s.unique()))
```

Out[39]:
```
B    2
C    4
D    2
E    2
F    2
G    4
dtype: int64
```

Series.apply()

By calling `.apply()` from a *Series*, it applies the function to **each element** of the *Series* **independently**.

For example:

In [40]:
```
df['C'].apply(lambda x: x**x)
```

Out[40]:
```
0      1.0
1      1.0
3     27.0
4    256.0
Name: C, dtype: float64
```

The above line applies the function $f(x) = x^x$ to every element x of `df['C']` .

This can be used to create **more complicated** filters!

Advanced filtering with `.apply()`

To do this, all we have to do is to create a function that returns `bool` values.

For example, we want to filter `df['B']` so that we keep entries with `30` days. First, we'll

In [41]:
```
df['B'].apply(lambda x: x.day == 30)
```

Out[41]:
```
0     True
1     True
3     True
4    False
Name: B, dtype: bool
```

The above is equivalent with:

```
# Write a function that returns a bool value based on
# the condition we want to filter the dataframe with
def has_30_days(x):
    # returns true if x has 30 days
    return x.day == 30

# Apply the function on column 'B'
df['B'].apply(has_30_days)
```

If we have created the function, all we have to do is to index the *DataFrame* with the result of the `.apply()`.

In [42]:
```
df[df['B'].apply(lambda x: x.day == 30)]
```

Out[42]:

	B	C	D	E	F	G
0	2019-03-30	0.0	3	test	foo	10
1	2019-03-30	1.0	3	train	foo	22
3	2019-03-30	3.0	3	train	foo	-8

Dealing with missing data

This is a very interesting topic, which we will revisit in more detail in a future tutorial.

In short there are a few easy ways we can quickly deal with missing data. The two main options are:

- Dropping missing data.
- Filling missing data.

Since *pandas* is built on top of *numpy*, missing data is represented with `np.nan` values. If they aren't, they'll have to be converted to `np.nan`.

Let's first download a sample *DataFrame* and fill it with missing values.

In [43]:
```
url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iri
data = np.genfromtxt(url, delimiter=',', dtype='float', usecols=[0,1,2,3]
data[np.random.randint(150, size=20), np.random.randint(4, size=20)] = np
data = pd.DataFrame(data, columns=['A', 'B', 'C', 'D'])
data.shape
```

Out[43]:
```
(150, 4)
```

This is 150×4 *DataFrame* with several missing values. How can we tell how many and where they are?

Inspecting missing values

This can be done with `.isna()` or `.isnull()`. What's the difference between the two? Nothing at all (here (https://datascience.stackexchange.com/a/37879/34269) in an explanation).

`DataFrame.isna()` checks every value one by one if it is `np.nan` or not. The only thing we have to do is aggregate the resulting *DataFrame*.

In [44]:
```
data.isna().any()  # checks if a column has at least one missing value or
```

Out[44]:
```
A    True
B    True
C    True
D    True
dtype: bool
```

In [45]:
```
data.isna().sum()  # how many missing values per column
```

Out[45]:
```
A    2
B    5
C    8
D    4
dtype: int64
```

In [46]:
```
data.isna().sum() / len(data) * 100  # percentage of values missing per c
```

Out[46]:
```
A    1.333333
B    3.333333
C    5.333333
D    2.666667
dtype: float64
```

Dropping missing values

There are two ways to drop a missing value:

- Drop its **row**.
- Drop its **column**.

Both can be accomplished through `dropna()`

In [47]:
```
tmp = data.dropna()         # drops rows with missing values
print(tmp.shape)
tmp = data.dropna(axis=1)   # drops columns with missing values
print(tmp.shape)
```

```
(131, 4)
(150, 0)
```

Note that these operations are **not inplace**! If we wanted to overwrite the original *DataFrame* we'd have to write:

```
data = data.dropna()
# or
data.dropna(inplace=True)
```

This method also offers many more parameters for

- dropping rows that have missing values **only in specific columns** (`subset`)
- dropping rows that have **multiple missing values** (more than a threshold `thres`)
- dropping rows (or columns) that have **all their values missing** (`how='all'`)

Filling missing values

This process is often referred to as **imputation**. In *pandas* it done with `.fillna()` and can be accomplished in two ways: either fill the whole *DataFrame* with a single value or fill the each column with a single value.

The first is the easiest to implement.

In [48]:
```
tmp = data.fillna(999)  # fills any missing value in the DataFrame with 9
print('Mean values for the original DataFrame:\n', data.mean())
print('\nMean values for the imputed DataFrame:\n', tmp.mean())
```

```
Mean values for the original DataFrame:
 A    5.846622
B    3.053793
C    3.709859
D    1.180822
dtype: float64

Mean values for the imputed DataFrame:
 A    19.088667
B    36.252000
C    56.792000
D    27.789333
dtype: float64
```

The second way is a bit more interesting. We'll first need to create a dictionary (or something equivalent) telling *pandas* which value to use for each column.

In [49]:
```
fill_values = {'A': -999, 'B':0, 'D': 999}  # note that we purposely ignc

tmp = data.fillna(fill_values)
print('Mean values for the original DataFrame:\n', data.mean())
print('\nMean values for the imputed DataFrame:\n', tmp.mean())
print('\nNumber of missing values of the original DataFrame:\n', data.isn
print('\nNumber of missing values of the imputed DataFrame:\n',tmp.isna()
```

```
Mean values for the original DataFrame:
 A    5.846622
B    3.053793
C    3.709859
D    1.180822
dtype: float64

Mean values for the imputed DataFrame:
 A   -7.551333
B    2.952000
C    3.709859
D   27.789333
dtype: float64

Number of missing values of the original DataFrame:
 A    2
B    5
C    8
D    4
dtype: int64

Number of missing values of the imputed DataFrame:
 A    0
B    0
C    8
D    0
dtype: int64
```

One interesting thing we can do is impute the missing values based on a statistic. For example, impute each missing value with its column's mean.

In [50]:
```
tmp = data.fillna(data.mean())
print('Mean values for the original DataFrame:\n', data.mean())
print('\nMean values for the imputed DataFrame:\n', tmp.mean())
```

```
Mean values for the original DataFrame:
 A    5.846622
B    3.053793
C    3.709859
D    1.180822
dtype: float64

Mean values for the imputed DataFrame:
 A    5.846622
B    3.053793
C    3.709859
D    1.180822
dtype: float64
```

Encoding data

Encoding is the process of converting columns containing alphanumeric values (`str`) to numeric ones (`int` or `float`).

This, too, will be covered in more detail in a later tutorial (*why is it necessary?, what ways there are? *what are the benefits of each?*). However, we'll show two easy ways this can be accomplished through *pandas*.

Label encoding

This essentially means mapping each `str` value to an `int` one. One way to do this is to create a dictionary that maps each `str` to an `int` and use the built-in method `.map()` .

In [51]:
```
mapping_dict = {'train': 0, 'test': 1}
df['E'].map(mapping_dict)  # this is NOT inplace
```

Out[51]:
```
0    1
1    0
3    0
4    0
Name: E, dtype: int64
```

Or we could use `.apply()` .

In [52]:
```
df['E'].apply(lambda x: mapping_dict[x])  # NOT inplace
```

Out[52]:
```
0    1
1    0
3    0
4    0
Name: E, dtype: int64
```

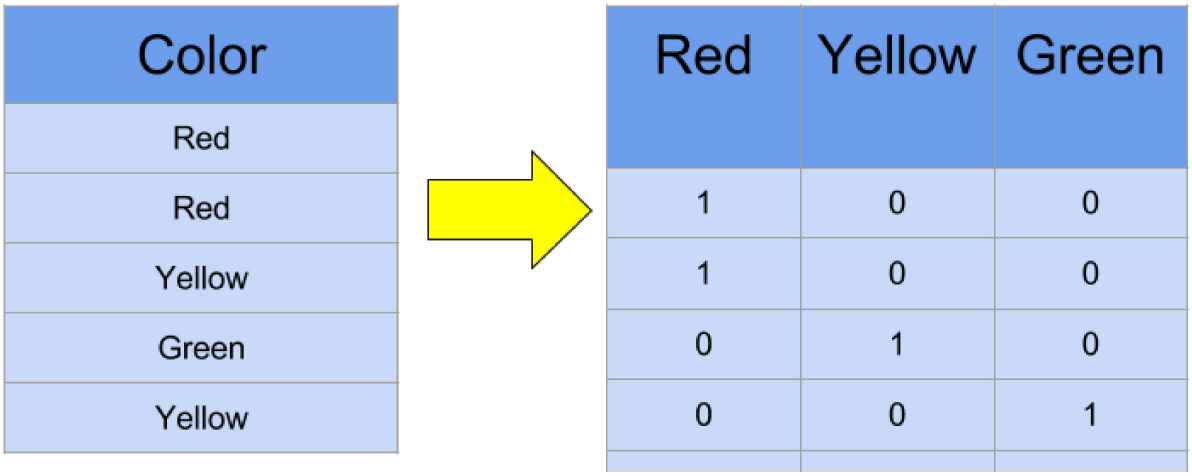
If we wanted to make the operations inplace we could simply write:

```
mapping_dict = {'train': 0, 'test': 1}

df['E'] = df['E'].map(mapping_dict)                          # using
map
# or
df['E'] = df['E'].apply(lambda x: mapping_dict_dict[x])   # using
apply
```

One-hot encoding

Also known as **dummy encoding**, this technique is a bit more complicated. To one-hot encode a column, we have to create as many new columns as there are unique values in the original column. Each of those represents one of the unique values. For each entry, we check the original value and set the corresponding new column to 1, while the rest are set to 0. An illustration of the process can be seen in the figure below.



The good news are that in *pandas* it is easier than it looks!

In [53]:
```
pd.get_dummies(df)  # only columns 'E' and 'F' need to be encoded
```

Out[53]:

	B	C	D	G	E_test	E_train	F_bar	F_foo
0	2019-03-30	0.0	3	10	1	0	0	1
1	2019-03-30	1.0	3	22	0	1	0	1
3	2019-03-30	3.0	3	-8	0	1	0	1
4	2019-03-31	4.0	-3	13	0	1	1	0

Again, this operation is **not** inplace.

Pivot tables

Pivot tables can provide important insight in the relationship between two or more variables.

Pandas actually offers to ways to generate pivot tables, one through a dedicated function `pd.pivot_table()` and one through the *DataFrame* method `.pivot()`. The first is **highly recommended** due to it allowing for the aggregation of duplicate values.

In [54]:
```
df2 = pd.DataFrame({'A': ['foo'] * 6 + ['bar'] * 4,
                    'B': ['one'] * 4 + ['two'] * 2 + ['one'] * 2 + ['two'
                    'C': ['small', 'large'] * 5,
                    'D': [1, 2, 2, 2, 3, 3, 4, 5, 6, 7]})

df2
```

Out[54]:

	A	B	C	D
0	foo	one	small	1
1	foo	one	large	2
2	foo	one	small	2
3	foo	one	large	2
4	foo	two	small	3
5	foo	two	large	3
6	bar	one	small	4
7	bar	one	large	5
8	bar	two	small	6
9	bar	two	large	7

A pivot table requires 3 things:

- `index` : A column so that its values can be set as the **rows** of the pivot table.
- `columns` : A column so that its values can be set as the **columns** of the pivot table.
- `values` : A column so that its values can be **aggregated** and placed into the grid defined by the rows and the columns of the pivot table.

In [55]:
```
pd.pivot_table(df2, index='A', columns='B', values='D')
```

Out[55]:

B	one	two
A		
bar	4.50	6.5
foo	1.75	3.0

The default aggregation function is `np.mean`. How is each position in the grid calculated?

The first element in the pivot table corresponds to `A == 'bar'` and `B == 'one'`. How many values do we have with this criteria?

In [56]:
```
df2[(df2['A'] == 'bar') & (df2['B'] == 'one')][['D']]
```

Out[56]:

	D
6	4
7	5

We said that by default *pandas* uses `np.mean` as its aggregator, so:

In [57]:
```
df2[(df2['A'] == 'bar') & (df2['B'] == 'one')]['D'].mean()
```

Out[57]:
```
4.5
```

Similarly, the second element in the pivot table has `A == 'bar'` and `B == 'two'`. So its value will be:

In [58]:
```
df2[(df2['A'] == 'bar') & (df2['B'] == 'two')]['D'].mean()
```

Out[58]:
```
6.5
```

Now, what if we want to change the aggregation function to something else, let's say `np.sum`.

In [59]:
```
pd.pivot_table(df2, index='A', columns='B', values='D', aggfunc=np.sum)
```

Out[59]:

B	one	two
A		
bar	9	13
foo	7	6

This simply sums the values of `'D'` that correspond to each position in the pivot table.

Another interesting choice for an aggregator is `len`. This will **count** the number of values in each position of the grid **instead of aggregating them**. This means the `values` argument is irrelevant when using `aggfunc=len`.

In [60]:
```
pd.pivot_table(df2, index='A', columns='B', values='D', aggfunc=len)
```

Out[60]:

B	one	two
A		
bar	2	2
foo	4	2

Creating custom functions for aggregation is also an option. For instance if we want to count the number of **unique values** per position:

In [61]:
```
pd.pivot_table(df2, index='A', columns='B', values='D', aggfunc=lambda x:
```

Out[61]:

A	B	one	two
bar		2	2
foo		2	1

Multi-index pivot tables are also an option but we won't go into any more detail.

In [62]:
```
pd.pivot_table(df2, index=['A', 'B'], columns='C', values='D', aggfunc=np
```

Out[62]:

A	B	C	large	small
bar	one		5	4
	two		7	6
foo	one		4	3
	two		3	3

Merging DataFrames

This is the action of combining two or more *DataFrames* into one. *Pandas* offers multiple ways of performing such a merger. Let's first create two *DataFrames* that share **only some** of their rows and columns.

In [63]:
```
df3 = pd.DataFrame({'A': ['df3'] * 4,
                    'B': ['df3'] * 4,
                    'C': ['df3'] * 4,
                    'D': ['df3'] * 4})
df3
```

Out[63]:

	A	B	C	D
0	df3	df3	df3	df3
1	df3	df3	df3	df3
2	df3	df3	df3	df3
3	df3	df3	df3	df3

In [64]:
```
df4 = pd.DataFrame({'B': ['df4'] * 4,
                    'D': ['df4'] * 4,
                    'F': ['df4'] * 4}, index=[2, 3, 6, 7])
df4
```

Out[64]:

	B	D	F
2	df4	df4	df4
3	df4	df4	df4
6	df4	df4	df4
7	df4	df4	df4

`df3` and `df4` have only one column and two rows in common.

Concatenation

Concatenating these two *DataFrames* is the simplest option and can be performed with `pd.concat()`. As we saw in the previous tutorial, there are two ways we can perform the concatenation:

- along the **rows** (`axis=0`) which would produce a *DataFrame* with $4 + 4 = 8$ rows
- along its **columns** (`axis=1`) which would produce a *DataFrame* with $4 + 3 = 7$ columns

Let's try the first.

In [65]:
```
pd.concat([df3, df4], sort=False)
```

Out[65]:

	A	B	C	D	F
0	df3	df3	df3	df3	NaN
1	df3	df3	df3	df3	NaN
2	df3	df3	df3	df3	NaN
3	df3	df3	df3	df3	NaN
2	NaN	df4	NaN	df4	df4
3	NaN	df4	NaN	df4	df4
6	NaN	df4	NaN	df4	df4
7	NaN	df4	NaN	df4	df4

This concatenation did append the rows of the second *DataFrame* under the first one, but the columns are out of alignment. Why is this?

This happens because *pandas* used the names of the columns to identify which columns to join. So `df4['B']` went under `df3['B']` and `df4['D']` went under `df3['D']`, but the rest of the columns don't match. The way *pandas* solved it is that it added column `'F'` to `df3` and

columns `'A'` and `'C'` to `df4` and filled them with `nan` values. Then it performed the merger as if both *DataFrames* were 4×5. This type of merger is called an **outer join** and it is the default for `pd.concat()` .

Also note that the rows with labels `2` and `3` are present two times in the *DataFrame*.

In [66]:
```
pd.concat([df3, df4], join='inner', sort=False)
```

Out[66]:

	B	D
0	df3	df3
1	df3	df3
2	df3	df3
3	df3	df3
2	df4	df4
3	df4	df4
6	df4	df4
7	df4	df4

The same things can be said about concatenating along the columns.

In [67]:
```
pd.concat([df3, df4], axis=1, sort=False)
```

Out[67]:

	A	B	C	D	B	D	F
0	df3	df3	df3	df3	NaN	NaN	NaN
1	df3	df3	df3	df3	NaN	NaN	NaN
2	df3	df3	df3	df3	df4	df4	df4
3	df3	df3	df3	df3	df4	df4	df4
6	NaN	NaN	NaN	NaN	df4	df4	df4
7	NaN	NaN	NaN	NaN	df4	df4	df4

Again, the rows that didn't exist (i.e. `6` and `7` in `df3` and `0` and `1` in `df4`) were created, the columns now have duplicate names (i.e. `'B'` and `'D'` appear twice) and all non-existing values were set to `nan` .

An inner join would look like this:

In [68]:
```
pd.concat([df3, df4], join='inner', axis=1, sort=False)
```

Out[68]:

	A	B	C	D	B	D	F
2	df3	df3	df3	df3	df4	df4	df4
3	df3	df3	df3	df3	df4	df4	df4

What if we just wanted to concatenate the *DataFrames*, though... like we did in *numpy* (i.e. join rows regardless their name). To do this we'd have to change the labels of the rows of the `df4` to match those of `df3`.

In [69]:
```
tmp = df4.copy()  # create a temporary DataFrame so that we don't overwri
tmp.index = df3.index  # change the index of df4 so that it's identical t
pd.concat([df3, tmp], axis=1, sort=False)
```

Out[69]:

	A	B	C	D	B	D	F
0	df3	df3	df3	df3	df4	df4	df4
1	df3	df3	df3	df3	df4	df4	df4
2	df3	df3	df3	df3	df4	df4	df4
3	df3	df3	df3	df3	df4	df4	df4

SQL-type joins

As we might have assumed from the previous step, *pandas* supports SQL-type joins.

The merger is performed on specific columns in both *DataFrames* (referred to as *keys*) or on the row labels (like we did before).

There are four types of joins:

- **outer**, which, as we saw before, uses the **union of the keys** of the two *DataFrames*.
  So the rows of the merger will be the rows that exist in both *DataFrames* (i.e. `2` and `3`), the rows that exist only in the first *DataFrame* (i.e. `0` and `1`) and the rows that exist only in the second *DataFrame* (i.e. `6` and `7`).
- **inner**, which, like before, uses **intersection of the keys** of the two *DataFrames*.
  Here the rows of the merger are only those existing in both *DataFrames* (i.e. `2` and `3`).
- **left**, which only keeps the keys of the **first** *DataFrame*.
  The rows will be the keys of the first *DataFrame* (i.e. `0`, `1`, `2` and `3`).
- **left**, which only keeps the keys of the **second** *DataFrame*.
  The rows will be the keys of the second *DataFrame* (i.e. `2`, `3`, `6` and `7`).

In all cases, by default, **all columns will be kept**. They will be, however, renamed if necessary so that there aren't any duplicate column names.

In [70]:
```
pd.merge(df3, df4, how='outer', left_index=True, right_index=True)   # the
                                                                     # to
```

Out[70]:

	A	B_x	C	D_x	B_y	D_y	F
0	df3	df3	df3	df3	NaN	NaN	NaN
1	df3	df3	df3	df3	NaN	NaN	NaN
2	df3	df3	df3	df3	df4	df4	df4
3	df3	df3	df3	df3	df4	df4	df4
6	NaN	NaN	NaN	NaN	df4	df4	df4
7	NaN	NaN	NaN	NaN	df4	df4	df4

In [71]:
```
pd.merge(df3, df4, how='inner', left_index=True, right_index=True)
```

Out[71]:

	A	B_x	C	D_x	B_y	D_y	F
2	df3	df3	df3	df3	df4	df4	df4
3	df3	df3	df3	df3	df4	df4	df4

In [72]:
```
pd.merge(df3, df4, how='left', left_index=True, right_index=True)
```

Out[72]:

	A	B_x	C	D_x	B_y	D_y	F
0	df3	df3	df3	df3	NaN	NaN	NaN
1	df3	df3	df3	df3	NaN	NaN	NaN
2	df3	df3	df3	df3	df4	df4	df4
3	df3	df3	df3	df3	df4	df4	df4

In [73]:
```
pd.merge(df3, df4, how='right', left_index=True, right_index=True)
```

Out[73]:

	A	B_x	C	D_x	B_y	D_y	F
2	df3	df3	df3	df3	df4	df4	df4
3	df3	df3	df3	df3	df4	df4	df4
6	NaN	NaN	NaN	NaN	df4	df4	df4
7	NaN	NaN	NaN	NaN	df4	df4	df4

"Group By" process

By “group by” we are referring to a process involving one or more of the following steps:

- **Splitting** the data into groups based on some criteria.
- **Applying** a function to each group independently.
  - **Aggregation**: compute a statistical summary of each group.
  - **Transformation**: perform an operation that alters the values in one or more groups.
  - **Filtration**: disregard some groups based on a group-wise computation.

- **Combining** the results into a data structure.

We'll use `df2` to illustrate this process

In [74]:
```
df2
```

Out[74]:

	A	B	C	D
0	foo	one	small	1
1	foo	one	large	2
2	foo	one	small	2
3	foo	one	large	2
4	foo	two	small	3
5	foo	two	large	3
6	bar	one	small	4
7	bar	one	large	5
8	bar	two	small	6
9	bar	two	large	7

Splitting the data

This step **partitions** the data into **subsets**, based on the values of a column.

In [75]:
```
grouped = df2.groupby(['A'])
```

Since `df2['A']` can take only too values (`'foo'` and `'bar'`), this is roughly equivalent to:

In [76]:
```
df2[df2['A'] == 'foo']
```

Out[76]:

	A	B	C	D
0	foo	one	small	1
1	foo	one	large	2
2	foo	one	small	2
3	foo	one	large	2
4	foo	two	small	3
5	foo	two	large	3

In [77]:
```
df2[df2['A'] == 'bar']
```

Out[77]:

	A	B	C	D
6	bar	one	small	4
7	bar	one	large	5
8	bar	two	small	6
9	bar	two	large	7

However, groupby **doesn't** actually perform the partitioning, it will do so when required in the next steps.

How can we access the groups?

In [78]:
```
grouped.groups
```

Out[78]:
```
{'bar': Int64Index([6, 7, 8, 9], dtype='int64'),
 'foo': Int64Index([0, 1, 2, 3, 4, 5], dtype='int64')}
```

This returns a dictionary with the unique values of `'A'` as its keys and the row indices that correspond to each key as its values.

If we know which key we want to use we can manually partition the data.

In [79]:
```
grouped.get_group('foo')
```

Out[79]:

	A	B	C	D
0	foo	one	small	1
1	foo	one	large	2
2	foo	one	small	2
3	foo	one	large	2
4	foo	two	small	3
5	foo	two	large	3

Applying functions

This step allows for the application of a function to each group independently. There are many types of operations we can perform here.

Aggregation

This involves generating a descriptive statistic for each of the groups.

In [80]:
```
grouped.agg(np.mean)  # the mean value of each column (only relevant for
```

Out[80]:

A	D
bar	5.500000
foo	2.166667

In [81]:
```
grouped.agg(len)  # how many samples does each group have
```

Out[81]:

A	B	C	D
bar	4	4	4
foo	6	6	6

We can even select a **different** aggregation function for each column.

In [82]:
```
grouped.agg({'B': len,                          # number of values in each
             'C': lambda x: len(x.unique()),    # unique values in each gro
             'D': np.sum})                      # sum the values of each gr
```

Out[82]:

A	B	C	D
bar	4	2	22
foo	6	2	13

Transformation

This involves changing some values in the data (each group's values are changed in a different manner). For example:

In [83]:
```
grouped.transform(lambda x: (x - x.min()) / (x.max() - x.min()))   # norma
grouped.transform(lambda x: (x - x.mean()) / x.std())              # stand
grouped.transform(lambda x: x.fillna(x.mean()))                    # repla
```

Out[83]:

	D
0	1
1	2
2	2
3	2
4	3
5	3
6	4
7	5
8	6
9	7

All the above operations are relevant only for column `'D'` (since it is the only containing numeric values) and are **not** performed inplace.

Filtering

This operation filters groups based on some condition.

In [84]:
```
grouped.filter(lambda x: x['D'].sum() > 15)  # keep only groups that have
```

Out[84]:

	A	B	C	D
6	bar	one	small	4
7	bar	one	large	5
8	bar	two	small	6
9	bar	two	large	7

Regular `.apply()`

All of the above three effects can be accomplished through `.apply()`.

In [85]:
```
# Aggregation:
grouped['D'].apply(np.sum)  # same as: grouped.apply(lambda x: x['D'].sum

# Transformation:
grouped['D'].apply(lambda x: (x - x.min()) / (x.max() - x.min()))
# equivalent with: grouped.apply(lambda x: (x['D'] - x['D'].min()) / (x['

# Filtering:
grouped.apply(lambda x: x if x['D'].sum() > 15 else None)
```

Out[85]:

		A	B	C	D
A					
bar	**6**	bar	one	small	4
	7	bar	one	large	5
	8	bar	two	small	6
	9	bar	two	large	7

The "group by" process can be done on multiple indices. However, we won't go more details about this.

In [86]:
```
df2.groupby(['A','B']).sum()  # roughly equivalent to the pivot_table we
```

Out[86]:

		D
A	**B**	
bar	**one**	9
	two	13
foo	**one**	7
	two	6

Shape manipulation

Unlike *numpy* arrays, *DataFrames* usually aren't made to be reshaped. Nevertheless, *pandas* offers support for stacking and unstacking.

In [87]:
```
# The stack() method "compresses" a level in the DataFrame's columns.
stk = df.stack()
print(stk)
```

```
0  B    2019-03-30 00:00:00
   C                      0
   D                      3
   E                   test
   F                    foo
   G                     10
1  B    2019-03-30 00:00:00
   C                      1
   D                      3
   E                  train
   F                    foo
   G                     22
3  B    2019-03-30 00:00:00
   C                      3
   D                      3
   E                  train
   F                    foo
   G                     -8
4  B    2019-03-31 00:00:00
   C                      4
   D                     -3
   E                  train
   F                    bar
   G                     13
dtype: object
```

In [88]:
```
# The inverse operation is unstack()
stk.unstack()
```

Out[88]:

		B	C	D	E	F	G
0	2019-03-30 00:00:00	0	3	test	foo	10
1	2019-03-30 00:00:00	1	3	train	foo	22
3	2019-03-30 00:00:00	3	3	train	foo	-8
4	2019-03-31 00:00:00	4	-3	train	bar	13

Input / output operations

The most common format associated with *DataFrames* is csv.

CSV

Writing a *DataFrame* to a csv file can be accomplished with a single line.

In [89]:
```
df.to_csv('tmp/my_dataframe.csv')  # writes df to file 'my_dataframe.csv'
```

`DataFrame.to_csv()` by default stores **both row and column labels**. Usually we don't want to write the row labels and sometimes we might not even want to write the column labels. This can be accomplished with the following arguments:

In [90]:
```
df.to_csv('tmp/my_dataframe.csv', header=False, index=False)
```

To load a csv into a *DataFrame* we can use `pd.read_csv()` .

In [91]:
```
tmp = pd.read_csv('tmp/my_dataframe.csv')
tmp
```

Out[91]:

	2019-03-30	0.0	3	test	foo	10
0	2019-03-30	1.0	3	train	foo	22
1	2019-03-30	3.0	3	train	foo	-8
2	2019-03-31	4.0	-3	train	bar	13

As you can see, by default, *pandas* uses the first line of the csv as its column names. If this isn't desirable, we can use the `header` argument.

In [92]:
```
tmp = pd.read_csv('tmp/my_dataframe.csv', header=None)
tmp
```

Out[92]:

	0	1	2	3	4	5
0	2019-03-30	0.0	3	test	foo	10
1	2019-03-30	1.0	3	train	foo	22
2	2019-03-30	3.0	3	train	foo	-8
3	2019-03-31	4.0	-3	train	bar	13

MS Excel

Pandas can read and write to excel files through two simple functions: `pd.read_excel(file.xlsx)` and `DataFrame.to_excel(file.xlsx)` . Note that this requires an extra library (`xlrd`)

Other options

Other options include pickle, json, SQL databases, clipboard, URLs and even integration with the google analytics API.

Exploratory Data Analysis

We've only scratched the surface of the capabilities of the *pandas* library. In order to get a better understanding of the library and how it's used, we'll attempt to perform an exploratory data analysis on the adult income dataset.

In [93]:
```
url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/adult/ad
data = pd.read_csv(url, header=None)
data.columns = ['age', 'workclass', 'fnlwgt', 'education', 'education-num
                'occupation', 'relationship', 'race', 'sex', 'capital-gai
                'hours-per-week', 'native-country', 'income']
```

The first thing we want to do is inspect the shape of the *DataFrame*.

In [94]:
```
data.shape
```

Out[94]:
```
(32561, 15)
```

Our data contains 32561 rows and 15 columns. If we take a look at the description (https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.names) of the dataset we see that it contains both continuous valued variables (age, working hours etc.) and categorical ones (sex, relationship etc.). When performing data analysis it is important to know what each variable represents.

The next thing we'll do is to look at a sample of the dataset.

In [95]:
```
data.head()
```

Out[95]:

	age	workclass	fnlwgt	education	education-num	marital-status	occupation	relationship	race	
0	39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	M
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	M
2	38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	M
3	53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black	M
4	28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black	Fem

For each variable we'll see what values it can take.

In [96]:
```
print('minimum:', data['age'].min())
print('maximum:', data['age'].max())
print('mean:   ', data['age'].mean())
```

```
minimum: 17
maximum: 90
mean:    38.58164675532078
```

`age` is a numeric variable that has a minimum value of 17 and a max of 90. While we can run any descriptive statistics on this variable, to have a complete perspective we must visualize it (see a later tutorial on how to do so).

In [97]:
```
data['workclass'].value_counts()
```

Out[97]:
```
 Private             22696
 Self-emp-not-inc     2541
 Local-gov            2093
 ?                    1836
 State-gov            1298
 Self-emp-inc         1116
 Federal-gov           960
 Without-pay            14
 Never-worked            7
Name: workclass, dtype: int64
```

Here we find our first occurrence of missing values. In this dataset, these are represented by question marks (`?`).

In [98]:
```
print('minimum:', data['fnlwgt'].min())
print('maximum:', data['fnlwgt'].max())
print('mean:   ', data['fnlwgt'].mean())
```

```
minimum: 12285
maximum: 1484705
mean:    189778.36651208502
```

This variable is continuous-valued and represents the demographics of the individual.

> Description of fnlwgt (final weight)

> The weights on the CPS files are controlled to independent estimates of the civilian noninstitutional population of the US. These are prepared monthly for us by Population Division here at the Census Bureau. We use 3 sets of controls. These are: 1. A single cell estimate of the population 16+ for each state. 2. Controls for Hispanic Origin by age and sex. 3. Controls by Race, age and sex.

> We use all three sets of controls in our weighting program and "rake" through them 6 times so that by the end we come back to all the controls we used.

> The term estimate refers to population totals derived from CPS by creating "weighted tallies" of any specified socio-economic characteristics of the population.

> People with similar demographic characteristics should have similar weights. There is one important caveat to remember about this statement. That is that since the CPS sample is actually a collection of 51 state samples, each with its own probability of selection, the statement only applies within state.

In [99]:
```
data['education'].value_counts()
```

Out[99]:
```
 HS-grad         10501
 Some-college     7291
 Bachelors        5355
 Masters          1723
 Assoc-voc        1382
 11th             1175
 Assoc-acdm       1067
 10th              933
 7th-8th           646
 Prof-school       576
 9th               514
 12th              433
 Doctorate         413
 5th-6th           333
 1st-4th           168
 Preschool          51
Name: education, dtype: int64
```

In [100]:
```
data['education-num'].value_counts()
```

Out[100]:
```
9     10501
10     7291
13     5355
14     1723
11     1382
7      1175
12     1067
6       933
4       646
15      576
5       514
8       433
16      413
3       333
2       168
1        51
Name: education-num, dtype: int64
```

The latter is simply an encoded version of the first.

In [101]:
```
data['marital-status'].value_counts()
```

Out[101]:
```
 Married-civ-spouse       14976
 Never-married            10683
 Divorced                  4443
 Separated                 1025
 Widowed                    993
 Married-spouse-absent      418
 Married-AF-spouse           23
Name: marital-status, dtype: int64
```

In [102]:
```
data['occupation'].value_counts()
```

Out[102]:
```
 Prof-specialty       4140
 Craft-repair         4099
 Exec-managerial      4066
 Adm-clerical         3770
 Sales                3650
 Other-service        3295
 Machine-op-inspct    2002
 ?                    1843
 Transport-moving     1597
 Handlers-cleaners    1370
 Farming-fishing       994
 Tech-support          928
 Protective-serv       649
 Priv-house-serv       149
 Armed-Forces            9
Name: occupation, dtype: int64
```

In [103]:
```
data['relationship'].value_counts()
```

Out[103]:
```
 Husband           13193
 Not-in-family      8305
 Own-child          5068
 Unmarried          3446
 Wife               1568
 Other-relative      981
Name: relationship, dtype: int64
```

In [104]:
```
data['race'].value_counts()
```

Out[104]:
```
 White                 27816
 Black                  3124
 Asian-Pac-Islander     1039
 Amer-Indian-Eskimo      311
 Other                   271
Name: race, dtype: int64
```

In [105]:
```
data['sex'].value_counts()
```

Out[105]:
```
 Male      21790
 Female    10771
Name: sex, dtype: int64
```

In [106]:
```
print(len(data[data['capital-gain'] == 0]))
print(len(data[data['capital-gain'] != 0]))
```

```
29849
2712
```

In [107]:
```
print(len(data[data['capital-loss'] == 0]))
print(len(data[data['capital-loss'] != 0]))
```

```
31042
1519
```

In [108]:
```
print('minimum:', data['hours-per-week'].min())
print('maximum:', data['hours-per-week'].max())
print('mean:   ', data['hours-per-week'].mean())
```

```
minimum: 1
maximum: 99
mean:    40.437455852092995
```

In [109]:
```
data['native-country'].value_counts()
```

Out[109]:
```
 United-States                 29170
 Mexico                          643
 ?                               583
 Philippines                     198
 Germany                         137
 Canada                          121
 Puerto-Rico                     114
 El-Salvador                     106
 India                           100
 Cuba                             95
 England                          90
 Jamaica                          81
 South                            80
 China                            75
 Italy                            73
 Dominican-Republic               70
 Vietnam                          67
 Guatemala                        64
 Japan                            62
 Poland                           60
 Columbia                         59
 Taiwan                           51
 Haiti                            44
 Iran                             43
 Portugal                         37
 Nicaragua                        34
 Peru                             31
 France                           29
 Greece                           29
 Ecuador                          28
 Ireland                          24
 Hong                             20
 Cambodia                         19
 Trinadad&Tobago                  19
 Thailand                         18
 Laos                             18
 Yugoslavia                       16
 Outlying-US(Guam-USVI-etc)       14
 Hungary                          13
 Honduras                         13
 Scotland                         12
 Holand-Netherlands                1
Name: native-country, dtype: int64
```

Data Preparation

Next, well look at several ways we might have to manipulate our data, including data cleaning, imputing and transforming

Because the unknown values are represented as question marks this dataset, we need to handle them.

Example: fill the occupation with the most frequent element

In [110]:
```
most_freq = data['occupation'].mode()[0]  # find the most common element

data['occupation'] = data['occupation'].apply(lambda x: most_freq if x ==
# the line above first keeps just the column that represents the occupati
# then it applies a function which checks if those values are question ma
# finally it replaces the original occupations with the new ones

data['occupation'].value_counts()
```

Out[110]:
```
 Prof-specialty       5983
 Craft-repair         4099
 Exec-managerial      4066
 Adm-clerical         3770
 Sales                3650
 Other-service        3295
 Machine-op-inspct    2002
 Transport-moving     1597
 Handlers-cleaners    1370
 Farming-fishing       994
 Tech-support          928
 Protective-serv       649
 Priv-house-serv       149
 Armed-Forces            9
Name: occupation, dtype: int64
```

Notice that all elements are proceeded by a whitespace? Can we remove it and clean our data?

In [111]:
```
print('Before cleaning: `{}`'.format(data.occupation[0]))
data.occupation = data['occupation'].apply(lambda x: x.strip())
print('After cleaning:  `{}`'.format(data.occupation[0]))
```

```
Before cleaning: ` Adm-clerical`
After cleaning:  `Adm-clerical`
```

Exercise 2

Fill the missing values of the DataFrame's `native-country` column with whatever strategy you wish.

Solution

This time we'll drop the rows containing missing values.

In [112]:
```
print('DataFrame length:', len(data))
print('missing:          ', len(data[data['native-country'] == ' ?']))

data = data.drop(data[data['native-country'] == ' ?'].index)

print('DataFrame length:', len(data))
print('missing:          ', len(data[data['native-country'] == ' ?']))
```

```
DataFrame length: 32561
missing:          583
DataFrame length: 31978
missing:          0
```

Finally, let's try to **encode** our data.

To illustrate how they would be performed in *pandas*: We will first encode the `education` variable preserving its sequential nature. Next, we will perform a custom encoding on the `marital-status` variable so that we keep only two categories (i.e. currently married and not). Finally, we will one-hot encode all the remaining categorical variables in the dataset.

First, `education`.

In [113]:
```
data['education'] = data['education'].apply(lambda x: x.strip())  # clean

# Create a dictionary mapping the categories to their encodings.
# This has to be done manually as the exact sequence has to be taken into
mappings = {'Preschool': 1, '1st-4th': 2, '5th-6th': 3, '7th-8th': 4, '9t
            '11th': 7, '12th': 8, 'HS-grad': 9, 'Some-college': 10, 'Asso
             'Bachelors': 13, 'Masters': 14, 'Prof-school': 15, 'Doctorat


data['education'] = data['education'].map(mappings)  # encode categorical
# another way to do this would be:  data.replace(mappings, inplace=True)
# another way this could be done would be through data.education.astype('
# this however would prevent us from choosing the mapping scheme
data['education'].value_counts()
```

Out[113]:
```
9     10368
10     7187
13     5210
14     1674
11     1366
7      1167
12     1055
6       921
4       627
15      559
5       506
8       417
16      390
3       318
2       163
1        50
Name: education, dtype: int64
```

Next, `marital-status`.

In [114]:
```
data['marital-status'] = np.where(data["marital-status"] == ' Married-civ
# the above function replaces ' Married-civ-spouse' with 1 and all the re

data['marital-status'].value_counts()
```

Out[114]:
```
0    17286
1    14692
Name: marital-status, dtype: int64
```

After this, we'll one-hot encode all rest categorical variables. Note that we haven't dealt with all missing values yet (in a real scenario we should).

In [115]:
```
print('Before one-hot encoding:', data.shape)

data = pd.get_dummies(data)  # one-hot encode all categorical variables

print('After one-hot encoding: ', data.shape)
```

```
Before one-hot encoding: (31978, 15)
After one-hot encoding:  (31978, 87)
```

Finally, we'll see how we can split numerical values to separate bins, in order to convert them to categorical. This time around we won't replace the numerical data but create a new variable.

In [116]:
```
data['age_categories'] = pd.cut(data.age, 3, labels=['young', 'middle age
data['age_categories'].value_counts()
```

Out[116]:
```
young          19556
middle aged    11280
old             1142
Name: age_categories, dtype: int64
```

When binning would usually want each bin to have the same number of samples. In order to do this we need to manually find there to cut each bin input the *cut points* instead of the number of bins we want. But we'll leave that up to you!

Bonus material:

Data wrangling:

- [extended data wrangling tutorial (http://nbviewer.jupyter.org/github/fonnesbeck/Bios8366/blob/master/notebooks/Section2_2-Data-Wrangling-with-Pandas.ipynb)]

Dealing with inconsistent text data

In [117]:
```
df6 = pd.DataFrame({'fname':['George', 'george ', 'GEORGIOS', 'Giorgos',
                    'sname':['Papadopoulos', 'alexakos ', 'Georgiou', 'ANT
                    'age': [46, 34, 75, 24, 54, 33]})
df6
```

Out[117]:

	fname	sname	age
0	George	Papadopoulos	46
1	george	alexakos	34
2	GEORGIOS	Georgiou	75
3	Giorgos	ANTONOPOULOS	24
4	Peter	Anastasiou	54
5	Petet	K	33

When looking at the example above, several inconsistencies become apparent. The first thing we want to do when dealing with strings is to convert them all to lowercase (or uppercase depending on preference) and remove preceding and succeeding whitespace.

In [118]:
```
def clean_text(text):
    text = text.strip()  # strip whitespace
    text = text.lower()  # convert to lowercase
    return text

df6['fname'] = df6['fname'].apply(clean_text)
df6['sname'] = df6['sname'].apply(clean_text)

# same could be done through a lambda function
# df.fname.apply(lambda x: x.strip().lower())

df6
```

Out[118]:

	fname	sname	age
0	george	papadopoulos	46
1	george	alexakos	34
2	georgios	georgiou	75
3	giorgos	antonopoulos	24
4	peter	anastasiou	54
5	petet	κ	33

Another problem originates from the way each name was entered. There are two ways to deal with this, one is to manually look and change errors, and the other is to compare the strings to find differences.

We are going to try the second, through the python package [fuzzywuzzy (https://github.com/seatgeek/fuzzywuzzy)](https://github.com/seatgeek/fuzzywuzzy).

In [119]:
```
from fuzzywuzzy import process
process.extract('george', df6['fname'], limit=None)
```

```
c:\users\thano\appdata\local\programs\python\python36\lib\site-package
s\fuzzywuzzy\fuzz.py:35: UserWarning: Using slow pure-python SequenceM
atcher. Install python-Levenshtein to remove this warning
  warnings.warn('Using slow pure-python SequenceMatcher. Install pytho
n-Levenshtein to remove this warning')
```

Out[119]:
```
[('george', 100, 0),
 ('george', 100, 1),
 ('georgios', 71, 2),
 ('giorgos', 62, 3),
 ('peter', 36, 4),
 ('petet', 36, 5)]
```

Fuzzywuzzy compares strings and outputs a score depending on how close they are. Let's replace the close ones:

In [120]:
```
def replace_matches_in_column(df, column, target_string, min_ratio=50):

    # find unique elements in specified column
    strings = df[column].unique()

    # see how close these elements are to the target string
    matches = process.extract(target_string, strings, limit=None)

    # keep only the closest ones
    close_matches = [matches[0] for matches in matches if matches[1] >= m

    # get the rows of all the close matches in our dataframe
    rows_with_matches = df[column].isin(close_matches)

    # replace all rows with close matches with the input matches
    df.loc[rows_with_matches, column] = target_string
```

In [121]:
```
replace_matches_in_column(df6, 'fname', 'george')
replace_matches_in_column(df6, 'fname', 'peter')

df6
```

Out[121]:

	fname	sname	age
0	george	papadopoulos	46
1	george	alexakos	34
2	george	georgiou	75
3	george	antonopoulos	24
4	peter	anastasiou	54
5	peter	κ	33

There, all clean! Note that the `min_ratio` we used is **very low**. We usually require much closer matches in order to replace.