# Fundamentals of Information Systems

## Python Programming (for Data Science)

### Master's Degree in Data Science

Gabriele Tolomei

gtolomei@math.unipd.it
University of Padua, Italy
2018/2019
November, 19 2018

# Lecture 8: I/O with `pandas`

# Overview

- Accessing data is a necessary first step for any data scientist.
- We are going to see how to perform data input/output operations using `pandas`.
- I/O might refer to: reading from/writing to text files (or other more efficient on-disk formats), accessing databases, interacting with network sources like web APIs, etc.
- We will be exploring each of those separately (although we will be focusing more on text files).

# Reading and Writing Text Files

# Parsing functions in `pandas` (1 of 2)

| Function | Description |
|---|---|
| `read_csv` | Load delimited data from a file, URL, or file-like object. Use comma as default delimiter |
| `read_table` | Load delimited data from a file, URL, or file-like object. Use tab (`'\t'`) as default delimiter |
| `read_fwf` | Read data in fixed-width column format (that is, no delimiters) |
| `read_clipboard` | Version of `read_table` that reads data from the clipboard. Useful for converting tables from web pages |
| `read_excel` | Read tabular data from an Excel XLS or XLSX file |
| `read_hdf` | Read pandas data from an HDF5 file |
| `read_html` | Read all tables found in the given HTML document |
| `read_json` | Read data from a JSON (JavaScript Object Notation) string representation |

# Parsing functions in `pandas` (2 of 2)

| | |
|---|---|
| `read_msgpack` | Read pandas data encoded using the MessagePack binary format |
| `read_pickle` | Read an arbitrary object stored in Python pickle format |
| `read_sas` | Read a SAS dataset stored in one of the SAS system's custom storage formats |
| `read_sql` | Read the results of a SQL query (using SQLAlchemy) as a pandas DataFrame. |
| `read_stata` | Read a dataset from Stata file format |
| `read_feather` | Read the Feather binary file format |

# Loading Data into `DataFrame` Objects

- All the functions listed above allow `pandas` to read tabular data as a `DataFrame` object.
- Among those, `read_csv` and `read_table` are by far the ones you'll likely use the most.

# Optional Arguments to `read_*` Functions

- **Indexing:** can treat one or more columns as the returned `DataFrame`, and whether to get column names from the file, the user, or not at all.
- **Type inference and data conversion:** this includes the user-defined value conversions and custom list of missing value markers.
- **Datetime parsing:** includes combining date and time information spread over multiple columns into a single column in the result.
- **Iterating:** support for iterating over chunks of very large files.
- **Unclean data issues:** skipping rows or a footer, comments, or other minor things like numeric data with thousands separated by commas.

# Too Many Optional Arguments

- Because of how messy data in the real world can be, some of the data loading functions (especially `read_csv`) have grown very complex over time.
- To avoid feeling ovewhelmed by the huge number of possible options, please refer to the online pandas documentation.
- Type inference is one of the more important features of these functions; that means you don't necessarily have to specify which columns are numeric, integer, boolean, or string.

# `read_csv`/`read_table`

- We will explore some of the most important I/O features provided by `pandas` using an example.
- To this end, we use a tabular data file located on a remote server.
- To check out how such a file looks like, just click here.
- Of course, you can save this file on your machine and load it locally from there with `pandas`.
- By default, data is assumed to be **tab-separated** (`'\t'`).

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

In [2]:
```
"""
Let's start with a real example on how to load a tabular data file using pandas.
"""
# Locate the dataset (in this case, we use a remote file located on an external server)
# Alternatively, you can save this file on your machine and load it locally from there.
url = 'https://raw.githubusercontent.com/justmarkham/DAT8/master/data/u.user'
# The first line of the file represents the header, and each field
# is separated by a pipe
"""
We specify the url where the data is located, the character used to separate fields ('|')
and the name of the column to use as row label (otherwise, RangeInteger will be used)
"""
users = pd.read_csv(url, sep='|', index_col='user_id')
print(users.head(10))
```

```
         age gender     occupation zip_code
user_id
1         24      M     technician    85711
2         53      F          other    94043
3         23      M         writer    32067
4         24      M     technician    43537
5         33      F          other    15213
6         42      M      executive    98101
7         57      M  administrator    91344
8         36      M  administrator    05201
9         29      M        student    01002
10        53      M         lawyer    90703
```

In [3]:
```
"""
Suppose we have stored the file on our local machine.
"""
path = './data/user_occupations.txt'
users = pd.read_csv(path, sep='|', index_col='user_id')
print(users.head(10))
```

```
         age gender     occupation zip_code
user_id
1         24      M     technician    85711
2         53      F          other    94043
3         23      M         writer    32067
4         24      M     technician    43537
5         33      F          other    15213
6         42      M      executive    98101
7         57      M  administrator    91344
8         36      M  administrator    05201
9         29      M        student    01002
10        53      M         lawyer    90703
```

In [4]:
```
"""
Suppose the file does not contain any header line. We can still load the file
telling pandas there is no header AND we can also provide pandas with a list
of names corresponding to the header we want to use.
"""
# This is the path to the same file yet without the header line
path_no_header = './data/user_occupations_no_header.txt'
# If the file does not contain the header as the first line
users = pd.read_csv(path_no_header, sep='|', header=None)
# Row and column indices fall back to the default RangeIndex (i.e., integers)
print(users.head(10))
```

```
    0   1  2              3      4
0   1  24  M     technician  85711
1   2  53  F          other  94043
2   3  23  M         writer  32067
3   4  24  M     technician  43537
4   5  33  F          other  15213
5   6  42  M     executive   98101
6   7  57  M  administrator  91344
7   8  36  M  administrator  05201
8   9  29  M        student  01002
9  10  53  M         lawyer  90703
```

In [5]:
```
# If the file does not contain the header as the first line AND we want to
# specify ourselves the names of the columns (and, possibly, the row index as well)
users = pd.read_csv(path_no_header, sep='|', header=None,
                    names=['user_id', 'age', 'gender', 'occupation', 'zip_code'],
                   index_col='user_id')

print(users.head(10))
```

```
         age gender     occupation zip_code
user_id
1         24      M     technician    85711
2         53      F          other    94043
3         23      M         writer    32067
4         24      M     technician    43537
5         33      F          other    15213
6         42      M      executive    98101
7         57      M  administrator    91344
8         36      M  administrator    05201
9         29      M        student    01002
10        53      M         lawyer    90703
```

In [6]:
```
# Sometimes, it may be useful to skip some records of the input file.
# Here, we skip the first, third and fourth (actual) record.
users_skip = pd.read_csv(path_no_header, sep='|', header=None,
                         names=['user_id', 'age', 'gender', 'occupation', 'zip_code'],
                        index_col='user_id',
                        skiprows=[0, 2, 3])

print(users_skip.head(10))
```

```
         age gender     occupation zip_code
user_id
2         53      F          other    94043
5         33      F          other    15213
6         42      M      executive    98101
7         57      M  administrator    91344
8         36      M  administrator    05201
9         29      M        student    01002
10        53      M         lawyer    90703
11        39      F          other    30329
12        28      F          other    06405
13        47      M       educator    29206
```

# Handling Missing Values (*NA* or *Not Available*)

- Missing data is usually either not present (i.e., empty string) or marked by some **sentinel** value.
- By default, `pandas` uses a set of commonly occurring sentinels, such as `None` and `NaN`.
- The `na_values` is used to customize sentinel values by adding to the default ones either a list or set of strings to consider missing values.

In [7]:
```
"""
Suppose we want to mark as NA any entry whose value is 'N/A'.
"""
# Load again the data with the option for handling missing values (na_values)
users = pd.read_csv(path, sep='|', index_col='user_id', na_values=['N/A'])
# Alternatively, we can define a dictionary of sentinels, i.e., a set for each column.
sentinels = {'age': ['inf', 'N/A'], 'zipcode': ['00000']}
users = pd.read_csv(path, sep='|', index_col='user_id', na_values=sentinels)
print(users.head(10))
```

```
         age gender     occupation zip_code
user_id
1         24      M     technician    85711
2         53      F          other    94043
3         23      M         writer    32067
4         24      M     technician    43537
5         33      F          other    15213
6         42      M      executive    98101
7         57      M  administrator    91344
8         36      M  administrator    05201
9         29      M        student    01002
10        53      M         lawyer    90703
```

# `read_csv`/`read_table` Function Arguments

| Argument | Description |
|---|---|
| `path` | String indicating filesystem location, URL, or file-like object |
| `sep` or `delimiter` | Character sequence or regular expression to use to split fields in each row |
| `header` | Row number to use as column names. Defaults to 0 (first row), but should be `None` if there is no header row |
| `index_col` | Column numbers or names to use as the row index in the result. Can be a single name/number or a list of them for a hierarchical index |
| `names` | List of column names for result, combine with `header=None` |
| `skiprows` | Number of rows at beginning of file to ignore or list of row numbers (starting from 0) to skip |
| `na_values` | Sequence of values to replace with NA |
| `comment` | Character or characters to split comments off the end of lines |
| `parse_dates` | Attempt to parse data to datetime; False by default. If True, will attempt to parse all columns. Otherwise can specify a list of column numbers or name to parse. If element of list is tuple or list, will combine multiple columns together and parse to date (for example if date/time split across two columns) |

# `read_csv`/`read_table` Function Arguments

| | |
|---|---|
| `keep_date_col` | If joining columns to parse date, keep the joined columns. Default False |
| `converters` | Dict containing column number of name mapping to functions. For example `{'foo': f}` would apply the function `f` to all values in the `'foo'` column |
| `dayfirst` | When parsing potentially ambiguous dates, treat as international format (e.g. 7/6/2012 -> June 7, 2012). Default False |
| `date_parser` | Function to use to parse dates |
| `nrows` | Number of rows to read from beginning of file |
| `iterator` | Return a TextParser object for reading file piecemeal |
| `chunksize` | For iteration, size of file chunks |
| `skip_footer` | Number of lines to ignore at end of file |
| `verbose` | Print various parser output information, like the number of missing values placed in non-numeric columns |
| `encoding` | Text encoding for unicode. For example `'utf-8'` for UTF-8 encoded text |
| `squeeze` | If the parsed data only contains one column return a Series |
| `thousands` | Separator for thousands, e.g. `','` or `'.'` |

# Reading Text Files in Pieces

- When processing very large files, you may only want to read in a small piece of a file or iterate through smaller chunks of the file.
- If we want to only read out a small number of rows (avoiding reading the entire file), specify that with `nrows`.

In [8]:
```
"""
Suppose we want to just read 100 records from our file.
"""
# Specify the number of rows to be read
users_100 = pd.read_csv(path, sep='|', index_col='user_id', nrows=100)
# Verify that we actually read that many rows
print("Number of observations (#rows) = {}".format(users_100.shape[0]))
users_100.head()
```

```
Number of observations (#rows) = 100
```

Out[8]:

| user_id | age | gender | occupation | zip_code |
|---|---|---|---|---|
| **1** | 24 | M | technician | 85711 |
| **2** | 53 | F | other | 94043 |
| **3** | 23 | M | writer | 32067 |
| **4** | 24 | M | technician | 43537 |
| **5** | 33 | F | other | 15213 |

In [9]:
```
"""
Let's reload the dataset from the remote file.
"""
users = pd.read_csv(url, sep='|', index_col='user_id')
print(users.head())
users.head()
```

```
         age gender  occupation zip_code
user_id
1         24      M  technician    85711
2         53      F       other    94043
3         23      M      writer    32067
4         24      M  technician    43537
5         33      F       other    15213
```

Out[9]:

| user_id | age | gender | occupation | zip_code |
|---|---|---|---|---|
| **1** | 24 | M | technician | 85711 |
| **2** | 53 | F | other | 94043 |
| **3** | 23 | M | writer | 32067 |
| **4** | 24 | M | technician | 43537 |
| **5** | 33 | F | other | 15213 |

In [10]:
```
"""
Let's print out some information about the data we just loaded.
"""
print("Number of observations (#rows) = {}".format(users.shape[0]))
print("Number of fields (#columns) = {}".format(users.shape[1]))
print("Column names = [{}]".format(", ".join([c for c in users.columns])))
print("The index (i.e., the labels) is:\n{}".format(users.index))
print("The data types of each column are:\n{}".format(users.dtypes))
```

```
Number of observations (#rows) = 943
Number of fields (#columns) = 4
Column names = [age, gender, occupation, zip_code]
The index (i.e., the labels) is:
Int64Index([  1,   2,   3,   4,   5,   6,   7,   8,   9,  10,
             ...
             934, 935, 936, 937, 938, 939, 940, 941, 942, 943],
            dtype='int64', name='user_id', length=943)
The data types of each column are:
age             int64
gender         object
occupation     object
zip_code       object
dtype: object
```

In [11]:
```
"""
Suppose we want to access a single column of the DataFrame.
"""
# Let's return the first 5 values of the 'occupation' column.
print(users['occupation'][:5]) # alternatively, use users['occupation'].head()
print()
# The same can be obtained using '.' notation
print(users.occupation[:5]) # alternatively, use users.occupation.head()
```

```
user_id
1    technician
2         other
3        writer
4    technician
5         other
Name: occupation, dtype: object

user_id
1    technician
2         other
3        writer
4    technician
5         other
Name: occupation, dtype: object
```

In [12]:
```
"""
Let's now create a deep copy of the loaded DataFrame 'users'.
Remember: assigning another name to the same DataFrame is simple a view.
For example, users_df = users makes users_df point to the same users. As such,
any change to the content of the DataFrame while working on users_df is reflected to users.
"""
# Make a deep copy of users
users_df = users.copy()
print(users_df.head())
```

```
         age gender  occupation zip_code
user_id
1         24      M  technician    85711
2         53      F       other    94043
3         23      M      writer    32067
4         24      M  technician    43537
5         33      F       other    15213
```

In [13]:
```
"""
Let's add an extra column to the DataFrame and populate this column
with some values (e.g., a series)
"""
# Suppose we want to add an extra column 'salary', which we randomly populate
# with values in the range [5000, 1000000]
np.random.seed(42) # Initialize internal state of the random number generator
BASE_SALARY = 5000
values = pd.Series(np.random.randint(995000, size=users_df.shape[0]) + BASE_SALARY)
print(values.head())
```

```
0    126958
1    676155
2    136932
3    370838
4    264178
dtype: int64
```

In [14]:
```
"""
Before we actually "join" the Series we have just created with our users DataFrame,
we need the index of both objects to be aligned. Otherwise, there won't be any salary
associated with the DataFrame row index 943, as the Series index is shifted by 1 w.r.t.
the index of our DataFrame. Let's specify the index when creating our salary values.
"""
np.random.seed(42) # Initialize internal state of the random number generator
BASE_SALARY = 5000
values = pd.Series(np.random.randint(995000, size=users_df.shape[0]) + BASE_SALARY,
                   index=users_df.index)
print(values.head())
```

```
user_id
1    126958
2    676155
3    136932
4    370838
5    264178
dtype: int64
```

In [15]:
```
# Create a new column on the users_df DataFrame and populate this with
# the Series we just created
users_df['salary'] = values
print(users_df.head())
```

```
         age gender  occupation zip_code  salary
user_id
1         24      M  technician    85711  126958
2         53      F       other    94043  676155
3         23      M      writer    32067  136932
4         24      M  technician    43537  370838
5         33      F       other    15213  264178
```

In [16]:
```
# We can access multiple columns of this new DataFrame as follows.
print("Occupation and Salary of the first 5 users:\n{}".
      format(users_df[['occupation', 'salary']].head()))
```

```
Occupation and Salary of the first 5 users:
         occupation  salary
user_id
1        technician  126958
2             other  676155
3            writer  136932
4        technician  370838
5             other  264178
```

In [17]:
```
"""
Wait! We might not want to associate a salary to each entry!
For example, you don't want to assign a salary to any user aged less than 18
or anyone who doesn't have a job or is a student.
Let's see what are the set of occupations.
"""
users_df['occupation'].unique()
```

Out[17]:
```
array(['technician', 'other', 'writer', 'executive', 'administrator',
       'student', 'lawyer', 'educator', 'scientist', 'entertainment',
       'programmer', 'librarian', 'homemaker', 'artist', 'engineer',
       'marketing', 'none', 'healthcare', 'retired', 'salesman', 'doctor'], dtype=object)
```

In [18]:
```
"""
Create a mask to assign a salary only to those users who are at least 18 AND
are not student nor unoccupied.
We therefore set salary to 0 for any of the users above
"""
mask = (users_df.age >= 18) & ~(users_df.occupation.isin(['student', 'none']))
# mask = (users_df.age >= 18) & (users_df.occupation != 'student') \
# & (users_df.occupation != 'none')
# mask = (users_df.age >= 18) & ~(users_df.occupation == 'student') \
# & ~(users_df.occupation == 'none')
```

In [19]:
```
users_df['salary'] = users_df['salary'].where(mask, 0)
# Alternatively
# users_df['salary'] = np.where(mask, users_df['salary'], 0)
users_df.head(10)
```

Out[19]:

| user_id | age | gender | occupation | zip_code | salary |
|---|---|---|---|---|---|
| **1** | 24 | M | technician | 85711 | 126958 |
| **2** | 53 | F | other | 94043 | 676155 |
| **3** | 23 | M | writer | 32067 | 136932 |
| **4** | 24 | M | technician | 43537 | 370838 |
| **5** | 33 | F | other | 15213 | 264178 |
| **6** | 42 | M | executive | 98101 | 649167 |
| **7** | 57 | M | administrator | 91344 | 115268 |
| **8** | 36 | M | administrator | 05201 | 737180 |
| **9** | 29 | M | student | 01002 | 0 |
| **10** | 53 | M | lawyer | 90703 | 142337 |

In [20]:
```
"""
Use integer slicing (special behavior to select rows)
"""
# Note that this integer slicing operator cannot be extended on both axis,
# as we did for 2-D numpy arrays. In other words, you cannot use the same
# syntax to slice over rows and columns with something like
# users2[i_start:i_stop, j_start:j_stop]
# In order to use integer slicing on BOTH axis as above, we need to use the .iloc method
print("First 7 rows of the DataFrame:\n{}".format(users_df[:7]))
```

```
First 7 rows of the DataFrame:
         age gender     occupation zip_code  salary
user_id
1         24      M     technician    85711  126958
2         53      F          other    94043  676155
3         23      M         writer    32067  136932
4         24      M     technician    43537  370838
5         33      F          other    15213  264178
6         42      M      executive    98101  649167
7         57      M  administrator    91344  115268
```

In [21]:
```
"""
Select all the users in the DataFrame whose salary is greater than 500k
"""
# This is a boolean mask which returns a Series containing either True or False
# corresponding to each entry index of the DataFrame depending on whether that entry
# has a salary which is greater than 500k or not.
mask = users_df.salary > 500000
print(mask.head(7))
print()
print("The list of first 5 users having salary greater than 500k is:\n{}"
      .format(users_df[mask].head()))
```

```
user_id
1    False
2     True
3    False
4    False
5    False
6     True
7    False
Name: salary, dtype: bool

The list of first 5 users having salary greater than 500k is:
         age gender     occupation zip_code  salary
user_id
2         53      F          other    94043  676155
6         42      M      executive    98101  649167
8         36      M  administrator    05201  737180
11        39      F          other    30329  526430
12        28      F          other    06405  959698
```

In [22]:
```
"""
Suppose I want to select only female users whose salary is greater than 500k
"""
mask = (users_df.salary > 500000) & (users_df.gender == 'F')
print(mask.head(7))
print()
print("The list of first 5 female users having salary greater than 500k is:\n{}"
      .format(users_df[mask].head()))
```

```
user_id
1    False
2     True
3    False
4    False
5    False
6    False
7    False
dtype: bool

The list of first 5 female users having salary greater than 500k is:
         age gender occupation zip_code  salary
user_id
2         53      F      other    94043  676155
11        39      F      other    30329  526430
12        28      F      other    06405  959698
15        49      F   educator    97301  917756
24        21      F     artist    94533  723315
```

In [23]:
```
"""
Let's use loc and iloc methods to index both axis (i.e., rows and columns)
using either index/column labels (loc) or integers (iloc).
"""
# Note that in this special case, index (row) labels are integers...
# In cases like this, loc falls back to work like .iloc
print("user_id: 1 and 4 (ROWS); gender, salary, zip_code (COLUMNS):\n{}"
      .format(users_df.loc[[1, 4], ['gender', 'salary', 'zip_code']]))
print()
print("user_id: 1 and 4 (ROWS); 2nd, 5th, 4th (COLUMNS):\n{}"
      .format(users_df.iloc[[0, 3], [1, 4, 3]]))
```

```
user_id: 1 and 4 (ROWS); gender, salary, zip_code (COLUMNS):
        gender  salary zip_code
user_id
1            M  126958    85711
4            M  370838    43537

user_id: 1 and 4 (ROWS); 2nd, 5th, 4th (COLUMNS):
        gender  salary zip_code
user_id
1            M  126958    85711
4            M  370838    43537
```

In [24]:
```
"""
Suppose we want to sort the DataFrame by age (ascending) and salary (descending)
"""
print(users_df.sort_values(by=['age', 'salary'], ascending=[True, False]).head())
```

```
         age gender occupation zip_code  salary
user_id
30         7      M    student    55436       0
471       10      M    student    77459       0
289       11      M       none    94619       0
142       13      M      other    48118       0
609       13      F    student    55106       0
```

In [25]:
```
"""
To make the above more meaningful, let's just consider only when salary is > 0
"""
print(users_df[users_df.salary > 0].sort_values(by=['age', 'salary'],
                                                ascending=[True, False]).head(10))
```

```
         age gender occupation zip_code  salary
user_id
620       18      F     writer    81648  719998
851       18      M      other    29646  522923
507       18      F     writer    28450  398002
925       18      F   salesman    49036  155159
859       18      F      other    06492   31790
747       19      M      other    93612  981535
601       19      F     artist    99687  755201
35        20      F  homemaker    42459  994436
596       20      M     artist    77073  749840
677       20      M      other    99835  748583
```

In [26]:
```
"""
Suppose we want to see what is the average salary of the users.
"""
# Let's first consider ALL the users (also those who have 0 salary)
print("The average salary across ALL the users is: {:.2f}"
      .format(users_df.salary.mean()))
# Let's now filter out from the mean computation any user whose salary is 0
print("The average salary across all working users is: {:.2f}"
      .format(users_df[users_df.salary > 0].salary.mean()))
```

```
The average salary across ALL the users is: 369708.87
The average salary across all working users is: 474333.96
```

In [27]:
```
"""
Let's see what is the median age of the users in our DataFrame.
"""
print("The median age across ALL the users is: {}"
      .format(users_df.age.median()))
```

```
The median age across ALL the users is: 31.0
```

In [28]:
```
"""
Let's see what happens if we call 'describe()' on this DataFrame
"""
print(users_df.describe()) # Notice, only numeric columns are part of the description!
```

```
                age         salary
count   943.000000     943.000000
mean     34.051962  369708.865323
std      12.192740  324238.271411
min       7.000000       0.000000
25%      25.000000   42366.500000
50%      31.000000  310062.000000
75%      43.000000  653225.000000
max      73.000000  994873.000000
```

In [29]:
```
# Let's try to include all the columns in the description
print(users_df.describe(include = "all"))
```

```
                age gender occupation zip_code         salary
count    943.000000    943        943      943     943.000000
unique          NaN      2         21      795            NaN
top             NaN      M    student    55414            NaN
freq            NaN    670        196        9            NaN
mean      34.051962    NaN        NaN      NaN  369708.865323
std       12.192740    NaN        NaN      NaN  324238.271411
min        7.000000    NaN        NaN      NaN       0.000000
25%       25.000000    NaN        NaN      NaN   42366.500000
50%       31.000000    NaN        NaN      NaN  310062.000000
75%       43.000000    NaN        NaN      NaN  653225.000000
max       73.000000    NaN        NaN      NaN  994873.000000
```

In [30]:
```
"""
Sometimes it is useful to know how the values of a particular attribute (i.e., column)
is distributed over the data instances that we have.
"""
# Let's first see how many unique occupations are on our dataset (already saw this above)
unique_occupations = users_df.occupation.unique()
print("There are {} unique occupation values, which are as follows:\n[{}]"
      .format(unique_occupations.shape[0],
              ", ".join([o.title() for o in np.sort(unique_occupations)])))
```

```
There are 21 unique occupation values, which are as follows:
[Administrator, Artist, Doctor, Educator, Engineer, Entertainment, Executive, Healthcare,
Homemaker, Lawyer, Librarian, Marketing, None, Other, Programmer, Retired, Salesman, Scien
tist, Student, Technician, Writer]
```

In [31]:
```
"""
Now let's see how many times each unique value of the 'occupation' column
appears across the dataset. In other words, we compute the frequency count (a.k.a. histogram)
of the 'occupation' attribute.
"""
print("Histogram of occupation values:\n{}"
      .format(pd.value_counts(users_df.occupation, sort=True)))
```

```
Histogram of occupation values:
student          196
other            105
educator          95
administrator     79
engineer          67
programmer        66
librarian         51
writer            45
executive         32
scientist         31
artist            28
technician        27
marketing         26
entertainment     18
healthcare        16
retired           14
salesman          12
lawyer            12
none               9
doctor             7
homemaker          7
Name: occupation, dtype: int64
```

# Working with other text formats

- Plain text files, such as `.csv` or `.tsv`, are not the only formats we might need work with.
- Other possible "text" formats can be: **JSON** (**J**ava**S**cript **O**bject **N**otation), **XML**/**HTML**, etc.
- We focus on **JSON** as this has become increasingly popular as a "standard" data exchange format (especially between distributed systems).

# What is JSON?

- JSON is strongly used as one of the standard formats for sending data via HTTP request between web browsers and servers.
- It is a much more free-form data format than a tabular text form like `csv`.
- The following is an example of a JSON **object**:

```
json_str = """
{"name": "Wes",
"places_lived": ["United States", "Spain", "Germany"],
"pet": null,
"siblings": [{"name": "Scott", "age": 29, "pets": ["Zeus
, "Zuko"]},
             {"name": "Katie", "age": 38,
              "pets": ["Sixes", "Stache", "Cisco"]}]
}
"""
```

# Similarity with pure Python

- JSON is very nearly valid Python code with the exception of its null value `null` and some other nuances.
- The basic types are **objects** (dicts), **arrays** (lists), **strings**, **numbers**, **booleans**, and **nulls**.
- All of the keys in an object must be strings.
- There are several Python modules for reading and writing JSON data; we'll use `json` here as it is built into the Python standard library.

In [32]:
```
"""
Let's import the json module.
"""
import json

# Assume we have the string json_obj above, representing a piece of JSON data
json_str = """
{"name": "Wes",
"places_lived": ["United States", "Spain", "Germany"],
"pet": null,
"siblings": [{"name": "Scott", "age": 29, "pets": ["Zeus", "Zuko"]},
           {"name": "Katie", "age": 38,
            "pets": ["Sixes", "Stache", "Cisco"]}]
}
"""
# We can load a JSON string into a JSON object using json.loads
json_obj = json.loads(json_str)
print(json_obj)
```

```
{'name': 'Wes', 'places_lived': ['United States', 'Spain', 'Germany'], 'pet': None, 'sibli
ngs': [{'name': 'Scott', 'age': 29, 'pets': ['Zeus', 'Zuko']}, {'name': 'Katie', 'age': 3
8, 'pets': ['Sixes', 'Stache', 'Cisco']}]}
```

In [33]:
```
"""
Suppose, instead, we want to load a JSON from an external file.
"""
# Let's have a JSON file (e.g., sample.json) stored somewhere on our local filesystem
json_filename = './data/sample.json'
with open(json_filename) as json_file:
    # We can load a JSON file into a JSON object using json.load
    json_obj = json.load(json_file)
    print(json_obj)
```

```
{'name': 'Wes', 'places_lived': ['United States', 'Spain', 'Germany'], 'pet': None, 'sibli
ngs': [{'name': 'Scott', 'age': 29, 'pets': ['Zeus', 'Zuko']}, {'name': 'Katie', 'age': 3
8, 'pets': ['Sixes', 'Stache', 'Cisco']}]}
```

In [ ]:
```
"""
Similarly, when we want to serialize out a JSON string or a JSON object
we can either use 'json.dumps(json_str, outfile)' in case of a JSON string
or json.dump(json_obj, outfile) in case of a JSON object
"""
# json_out_filename = './data/sample_out.json'
# with open(json_out_filename, 'w') as json_out_file:
#     json.dumps(json_str, json_outfile) # json_str is a (JSON) string object
#     json.dump(json_obj, json_outfile) # json_obj is a JSON object
```

In [35]:
```
"""
How to convert a JSON object into a pandas DataFrame.
For example, we can pass a list of JSON objects to the DataFrame constructor
and select a subset of the data fields.
"""
siblings = pd.DataFrame(json_obj['siblings'], columns=['name', 'age'])
print(siblings.head())
```

```
    name  age
0  Scott   29
1  Katie   38
```

In [36]:
```
"""
Alternatively, there is also a 'read_json' method that pandas provides, which allows us
to directly load a JSON file (or a JSON string) into a pandas DataFrame (i.e., without
the need of first loading it into a JSON object).
The default options for pandas.read_json assume that each object in the JSON array
is a row in the table.
"""
df = pd.read_json('./data/sample_df.json')
print(df.head())
```

```
     color value
0      red  #f00
1    green  #0f0
2     blue  #00f
3     cyan  #0ff
4  magenta  #f0f
```

# Working with Binary File Formats

- One of the easiest ways to efficiently work with data in **binary format** is using Python's built-in `pickle` serialization/de-serialization.
- This module essentially provides the ability to serialize (de-serialize) **any** Python object to (from) disk.
- Serialize means writing out a binary representation of an object, whilst de-serialize is the inverse operation.
- `pandas` objects all have a `to_pickle` method which writes data to disk in `pickle` format, whereas there exists a `pandas.read_pickle` method to deserialize objects back from disk.

In [37]:
```
"""
Suppose we want to serialize out to disk (in binary format) a DataFrame,
which we previously loaded from a text file.
"""
# Let's use again the same text file as above
path = './data/user_occupations.txt'
users = pd.read_csv(path, sep='|', index_col='user_id')
print(users.head())
```

```
         age gender  occupation zip_code
user_id
1         24      M  technician    85711
2         53      F       other    94043
3         23      M      writer    32067
4         24      M  technician    43537
5         33      F       other    15213
```

In [38]:
```
"""
Suppose we want to serialize out to (de-serialize in from) disk
a DataFrame (in binary format), which we previously loaded from a text file.
"""
# Let's serialize this DataFrame object out to a binary file on disk
binary_outfilename = './data/user_occupations.pickle'
# the same as pickle.dump(users, open(binary_outfilename, 'wb'))
users.to_pickle(binary_outfilename)
# Let's de-serialize the binary representation of the object back
users_from_binary = pd.read_pickle(binary_outfilename)
# This has given us back the DataFrame object we previously serialized
print(users_from_binary.head())
```

```
         age gender  occupation zip_code
user_id
1         24      M  technician    85711
2         53      F       other    94043
3         23      M      writer    32067
4         24      M  technician    43537
5         33      F       other    15213
```

## `pandas` Supports Many Binary Formats

- `pandas` has built-in support for two more binary data formats: HDF5 and MessagePack.
- I encourage you to explore different file formats to see how fast they are and how well they work for your task.