Course › Modul... › Solutio... › Sample...

## Sample solutions

# Tidy data and the Pandas module

This notebook accompanies Topic 7, which is about "tidying data," or cleaning up tabular data for analysis purposes. It also introduces one of the most important Python modules for data analysis: Pandas! (not the bear)

> *Note*: All parts are included in this single notebook.

# Part 0: Getting the data

Before beginning, you'll need to download several files containing the data for the exercises below.

**Exercise 0** (ungraded). Run the code cell below to download the data. (This code will check if each dataset has already been downloaded and, if so, will avoid re-downloading it.)

In [1]:
```
import requests
import os
import hashlib
import io

def download(file, url_suffix=None, checksum=None):
    if url_suffix is None:
        url_suffix = file

    if not os.path.exists(file):
        url = 'https://cse6040.gatech.edu/datasets/{}'.format(url_suffix)
        print("Downloading: {} ...".format(url))
        r = requests.get(url)
        with open(file, 'w', encoding=r.encoding) as f:
            f.write(r.text)

    if checksum is not None:
        with io.open(file, 'r', encoding='utf-8', errors='replace') as f:
            body = f.read()
            body_checksum = hashlib.md5(body.encode('utf-8')).hexdigest()
            assert body_checksum == checksum, \
                "Downloaded file '{}' has incorrect checksum: '{}' instead of '{}'".format(file, body_checksum, che
cksum)

    print("'{}' is ready!".format(file))

datasets = {'iris.csv': 'd1175c032e1042bec7f974c91e4a65ae',
            'table1.csv': '556ffe73363752488d6b41462f5ff3c9',
            'table2.csv': '16e04efbc7122e515f7a81a3361e6b87',
            'table3.csv': '531d13889f191d6c07c27c3c7ea035ff',
            'table4a.csv': '3c0bbecb40c6958df33a1f9aa5629a80',
            'table4b.csv': '8484bcdf07b50a7e0932099daa72a93d',
            'who.csv': '59fed6bbce66349bf00244b550a93544',
            'who2_soln.csv': 'f6d4875feea9d6fca82ae7f87f760f44',
            'who3_soln.csv': 'fba14f1e088d871e4407f5f737cfbc06'}

for filename, checksum in datasets.items():
    download(filename, url_suffix='tidy/{}'.format(filename), checksum=checksum)

print("\n(All data appears to be ready.)")
```

```
'table4b.csv' is ready!
'who2_soln.csv' is ready!
'who.csv' is ready!
'table4a.csv' is ready!
'iris.csv' is ready!
'table1.csv' is ready!
'table3.csv' is ready!
'who3_soln.csv' is ready!
'table2.csv' is ready!

(All data appears to be ready.)
```

# Part 1: Tidy data

The overall topic for this lab is what we'll refer to as representing data *relationally*. The topic of this part is a specific type of relational representation sometimes referred to as the *tidy* (as opposed to *untidy* or *messy*) form. The concept of tidy data was developed by Hadley Wickham (http://hadley.nz/), a statistician and R programming maestro. Much of this lab is based on his tutorial materials (see below).

If you know SQL (https://en.wikipedia.org/wiki/SQL), then you are already familiar with relational data representations. However, we might discuss it a little differently from the way you may have encountered the subject previously. The main reason is our overall goal in the class: to build data *analysis* pipelines. If our end goal is analysis, then we often want to extract or prepare data in a way that makes analysis easier.

You may find it helpful to also refer to the original materials on which this lab is based:

- Wickham's R tutorial on making data tidy: http://r4ds.had.co.nz/tidy-data.html (http://r4ds.had.co.nz/tidy-data.html)
- The slides from a talk by Wickham on the concept: http://vita.had.co.nz/papers/tidy-data-pres.pdf (http://vita.had.co.nz/papers/tidy-data-pres.pdf)
- Wickham's more theoretical paper of "tidy" vs. "untidy" data: http://www.jstatsoft.org/v59/i10/paper (http://www.jstatsoft.org/v59/i10/paper)

---

## What is tidy data?

To build your intuition, consider the following data set collected from a survey or study.

**Representation 1.** Two-way contigency table (https://en.wikipedia.org/wiki/Contingency_table).

| | Pregnant | Not pregnant |
|---|---|---|
| **Male** | 0 | 5 |
| **Female** | 1 | 4 |

**Representation 2.** Observation list or "data frame."

| Gender | Pregnant | Count |
|---|---|---|
| Male | Yes | 0 |
| Male | No | 5 |
| Female | Yes | 1 |
| Female | No | 4 |

These are two entirely equivalent ways of representing the same data. However, each may be suited to a particular task.

For instance, Representation 1 is a typical input format for statistical routines that implement Pearson's $\chi^2$-test, which can check for independence between factors. (Are gender and pregnancy status independent?) By contrast, Representation 2 might be better suited to regression. (Can you predict relative counts from gender and pregnancy status?)

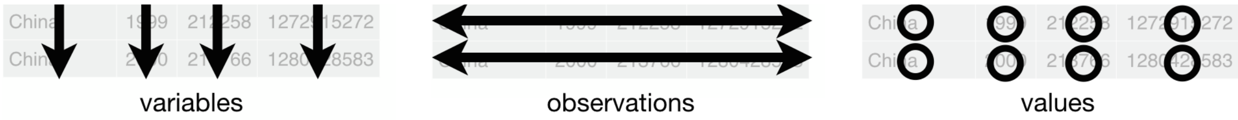
While Representation 1 has its uses (http://simplystatistics.org/2016/02/17/non-tidy-data/), Wickham argues that Representation 2 is often the cleaner and more general way to supply data to a wide variety of statistical analysis and visualization tasks. He refers to Representation 2 as *tidy* and Representation 1 as *untidy* or *messy*.

> The term "messy" is, as Wickham states, not intended to be perjorative since "messy" representations may be exactly the right ones for particular analysis tasks, as noted above.

**Definition: Tidy datasets.** More specifically, Wickham defines a tidy data set as one that can be organized into a 2-D table such that

1. each column represents a *variable*;
2. each row represents an *observation*;
3. each entry of the table represents a single *value*, which may come from either categorical (discrete) or continuous spaces.

Here is a visual schematic of this definition, taken from another source (http://r4ds.had.co.nz/images/tidy-1.png):




This definition appeals to a statistician's intuitive idea of data he or she wishes to analyze. It is also consistent with tasks that seek to establish a functional relationship between some response (output) variable from one or more independent variables.

> A computer scientist with a machine learning outlook might refer to columns as *features* and rows as *data points*, especially when all values are numerical (ordinal or continuous).

**Definition: Tibbles.** Here's one more bit of terminology: if a table is tidy, we will call it a *tidy table*, or *tibble*, for short.

## Part 2: Tidy Basics and Pandas

In Python, the Pandas (http://pandas.pydata.org/) module is a convenient way to store tibbles. If you know R (http://r-project.org), you will see that the design and API of Pandas's data frames derives from R's data frames (https://stat.ethz.ch/R-manual/R-devel/library/base/html/data.frame.html).

In this part of this notebook, let's look at how Pandas works and can help us store Tidy data.

You may find this introduction to the Pandas module's data structures useful for reference:

- https://pandas.pydata.org/pandas-docs/stable/dsintro.html (https://pandas.pydata.org/pandas-docs/stable/dsintro.html)

Consider the famous Iris data set (https://en.wikipedia.org/wiki/Iris_flower_data_set). It consists of 50 samples from each of three species of Iris (*Iris setosa*, *Iris virginica*, and *Iris versicolor*). Four features were measured from each sample: the lengths and the widths of the sepals (https://en.wikipedia.org/wiki/Sepal) and petals (https://en.wikipedia.org/wiki/Petal).

The following code uses Pandas to read and represent this data in a Pandas data frame object, stored in a variable named `irises`.

In [2]:
```
# Some modules you'll need in this part
import pandas as pd
from io import StringIO
from IPython.display import display

# Ignore this line. It will be used later.
SAVE_APPLY = getattr(pd.DataFrame, 'apply')

irises = pd.read_csv('iris.csv')
print("=== Iris data set: {} rows x {} columns. ===".format(irises.shape[0], irises.shape[1]))
display (irises.head())
```

```
=== Iris data set: 150 rows x 5 columns. ===
```

| | sepal length | sepal width | petal length | petal width | species |
|---|---|---|---|---|---|
| **0** | 5.1 | 3.5 | 1.4 | 0.2 | Iris-setosa |
| **1** | 4.9 | 3.0 | 1.4 | 0.2 | Iris-setosa |
| **2** | 4.7 | 3.2 | 1.3 | 0.2 | Iris-setosa |
| **3** | 4.6 | 3.1 | 1.5 | 0.2 | Iris-setosa |
| **4** | 5.0 | 3.6 | 1.4 | 0.2 | Iris-setosa |

In a Pandas data frame, every column has a name (stored as a string) and all values within the column must have the same primitive type. This fact makes columns different from, for instance, lists.

In addition, every row has a special column, called the data frame's *index*. (Try printing `irises.index`.) Any particular index value serves as a name for its row; these index values are usually integers but can be more complex types, like tuples.

In [3]:
```
print(irises.index)
```

```
RangeIndex(start=0, stop=150, step=1)
```

Separate from the index values (row names), you can also refer to rows by their integer offset from the top, where the first row has an offset of 0 and the last row has an offset of `n-1` if the data frame has `n` rows. You'll see that in action in Exercise 1, below.

**Exercise 1** (ungraded). Run the following commands to understand what each one does. If it's not obvious, try reading the Pandas documentation (http://pandas.pydata.org/) or going online to get more information.

```
irises.describe()
irises['sepal length'].head()
irises[["sepal length", "petal width"]].head()
irises.iloc[5:10]
irises[irises["sepal length"] > 5.0]
irises["sepal length"].max()
irises['species'].unique()
irises.sort_values(by="sepal length", ascending=False).head(1)
irises.sort_values(by="sepal length", ascending=False).iloc[5:10]
irises.sort_values(by="sepal length", ascending=False).loc[5:10]
irises['x'] = 3.14
irises.rename(columns={'species': 'type'})
del irises['x']
```

In [4]:

```
### BEGIN SOLUTION
print("\n=== `irises.describe()`: Prints summary statistics ===\n\n{}".format(irises.describe()))
print("\n=== `irises['sepal length'].head()`: Dumps the first few rows of a given column ===\n\n{}".format(irises[
'sepal length'].head()))
print('\n=== `irises[["sepal length", "petal width"]].head()`: Dumps the first few rows of several specific columns
 ===\n\n{}'.format(irises[["sepal length", "petal width"]].head()))
print("\n=== `irises.iloc[5:10]`: Selects rows at a certain integer offset and range ===\n\n{}".format(irises.iloc[
5:10]))
print('\n=== `irises[irises["sepal length"] > 5.0]`: Selects the subset of rows satisfying some condition (here, wh
ere sepal length is strictly more than 5) ===\n\n{}'.format(irises[irises["sepal length"] > 5.0]))
print('\n=== `irises["sepal length"].max()`: Returns the largest value of a given column ===\n\n{}'.format(irises[
"sepal length"].max()))
print("\n=== `irises['species'].unique()`: Returns a list of unique values in a given column ===\n\n{}".format(iris
es['species'].unique()))
print('\n=== `irises.sort_values(by="sepal length", ascending=False).head(1)`: Returns the observation with the lon
gest sepal length ===\n\n{}'.format(irises.sort_values(by="sepal length", ascending=False).head(1)))
print('\n=== `irises.sort_values(by="sepal length", ascending=False).iloc[5:10]`: Returns the observations whose ra
nks, in highest sepal length, are 5-9 inclusive ===\n\n{}'.format(irises.sort_values(by="sepal length", ascending=F
alse).iloc[5:10]))
print('\n=== `irises.sort_values(by="sepal length", ascending=False).loc[5:10]`: Returns the observations between t
he one whose row ID is 5 and the one that is 10, in order of sepal-length, 5 and 10 are inclusive ===\n\n{}'.format
(irises.sort_values(by="sepal length", ascending=False).loc[5:10]))

irises['x'] = 3.14
print("\n=== `irises['x'] = 3.14`: Creates a new column (variable) named ‘x’ and sets all values in column = 3.14 =
==\n\n{}".format(irises.head()))

irises2 = irises.rename(columns={'species': 'type'})
print("\n=== irises.rename(columns={{'species': 'type'}}): Change the name of a column (variable) ===\n\n{}".format
(irises2))

del irises['x']
print("\n=== `del irises['x']`: Removes a column ===\n\n{}".format(irises.head()))

### END SOLUTION
```

```
=== `irises.describe()`: Prints summary statistics ===

       sepal length  sepal width  petal length  petal width
count    150.000000   150.000000    150.000000   150.000000
mean       5.843333     3.057333      3.758000     1.199333
std        0.828066     0.435866      1.765298     0.762238
min        4.300000     2.000000      1.000000     0.100000
25%        5.100000     2.800000      1.600000     0.300000
50%        5.800000     3.000000      4.350000     1.300000
75%        6.400000     3.300000      5.100000     1.800000
max        7.900000     4.400000      6.900000     2.500000

=== `irises['sepal length'].head()`: Dumps the first few rows of a given column ===

0    5.1
1    4.9
2    4.7
3    4.6
4    5.0
Name: sepal length, dtype: float64

=== `irises[["sepal length", "petal width"]].head()`: Dumps the first few rows of several specific columns ===

   sepal length  petal width
0           5.1          0.2
1           4.9          0.2
2           4.7          0.2
3           4.6          0.2
4           5.0          0.2

=== `irises.iloc[5:10]`: Selects rows at a certain integer offset and range ===

   sepal length  sepal width  petal length  petal width      species
5           5.4          3.9           1.7          0.4  Iris-setosa
6           4.6          3.4           1.4          0.3  Iris-setosa
```

```
7          5.0          3.4          1.5          0.2  Iris-setosa
8          4.4          2.9          1.4          0.2  Iris-setosa
9          4.9          3.1          1.5          0.1  Iris-setosa

=== `irises[irises["sepal length"] > 5.0]`: Selects the subset of rows satisfying some condition (here, where sepa
l length is strictly more than 5) ===

     sepal length  sepal width  petal length  petal width          species
0             5.1          3.5           1.4          0.2      Iris-setosa
5             5.4          3.9           1.7          0.4      Iris-setosa
10            5.4          3.7           1.5          0.2      Iris-setosa
14            5.8          4.0           1.2          0.2      Iris-setosa
15            5.7          4.4           1.5          0.4      Iris-setosa
16            5.4          3.9           1.3          0.4      Iris-setosa
17            5.1          3.5           1.4          0.3      Iris-setosa
18            5.7          3.8           1.7          0.3      Iris-setosa
19            5.1          3.8           1.5          0.3      Iris-setosa
20            5.4          3.4           1.7          0.2      Iris-setosa
21            5.1          3.7           1.5          0.4      Iris-setosa
23            5.1          3.3           1.7          0.5      Iris-setosa
27            5.2          3.5           1.5          0.2      Iris-setosa
28            5.2          3.4           1.4          0.2      Iris-setosa
31            5.4          3.4           1.5          0.4      Iris-setosa
32            5.2          4.1           1.5          0.1      Iris-setosa
33            5.5          4.2           1.4          0.2      Iris-setosa
36            5.5          3.5           1.3          0.2      Iris-setosa
39            5.1          3.4           1.5          0.2      Iris-setosa
44            5.1          3.8           1.9          0.4      Iris-setosa
46            5.1          3.8           1.6          0.2      Iris-setosa
48            5.3          3.7           1.5          0.2      Iris-setosa
50            7.0          3.2           4.7          1.4  Iris-versicolor
51            6.4          3.2           4.5          1.5  Iris-versicolor
52            6.9          3.1           4.9          1.5  Iris-versicolor
53            5.5          2.3           4.0          1.3  Iris-versicolor
54            6.5          2.8           4.6          1.5  Iris-versicolor
55            5.7          2.8           4.5          1.3  Iris-versicolor
56            6.3          3.3           4.7          1.6  Iris-versicolor
58            6.6          2.9           4.6          1.3  Iris-versicolor
..            ...          ...           ...          ...              ...
120           6.9          3.2           5.7          2.3   Iris-virginica
121           5.6          2.8           4.9          2.0   Iris-virginica
122           7.7          2.8           6.7          2.0   Iris-virginica
123           6.3          2.7           4.9          1.8   Iris-virginica
124           6.7          3.3           5.7          2.1   Iris-virginica
125           7.2          3.2           6.0          1.8   Iris-virginica
126           6.2          2.8           4.8          1.8   Iris-virginica
127           6.1          3.0           4.9          1.8   Iris-virginica
128           6.4          2.8           5.6          2.1   Iris-virginica
129           7.2          3.0           5.8          1.6   Iris-virginica
130           7.4          2.8           6.1          1.9   Iris-virginica
131           7.9          3.8           6.4          2.0   Iris-virginica
132           6.4          2.8           5.6          2.2   Iris-virginica
133           6.3          2.8           5.1          1.5   Iris-virginica
134           6.1          2.6           5.6          1.4   Iris-virginica
135           7.7          3.0           6.1          2.3   Iris-virginica
136           6.3          3.4           5.6          2.4   Iris-virginica
137           6.4          3.1           5.5          1.8   Iris-virginica
138           6.0          3.0           4.8          1.8   Iris-virginica
139           6.9          3.1           5.4          2.1   Iris-virginica
140           6.7          3.1           5.6          2.4   Iris-virginica
141           6.9          3.1           5.1          2.3   Iris-virginica
142           5.8          2.7           5.1          1.9   Iris-virginica
143           6.8          3.2           5.9          2.3   Iris-virginica
144           6.7          3.3           5.7          2.5   Iris-virginica
145           6.7          3.0           5.2          2.3   Iris-virginica
146           6.3          2.5           5.0          1.9   Iris-virginica
147           6.5          3.0           5.2          2.0   Iris-virginica
148           6.2          3.4           5.4          2.3   Iris-virginica
149           5.9          3.0           5.1          1.8   Iris-virginica

[118 rows x 5 columns]

=== `irises["sepal length"].max()`: Returns the largest value of a given column ===

7.9

=== `irises['species'].unique()`: Returns a list of unique values in a given column ===

['Iris-setosa' 'Iris-versicolor' 'Iris-virginica']

=== `irises.sort_values(by="sepal length", ascending=False).head(1)`: Returns the observation with the longest sep
al length ===

     sepal length  sepal width  petal length  petal width         species
131           7.9          3.8           6.4          2.0  Iris-virginica

=== `irises.sort_values(by="sepal length", ascending=False).iloc[5:10]`: Returns the observations whose ranks, in
highest sepal length, are 5-9 inclusive ===
```

```
     sepal length  sepal width  petal length  petal width         species
105           7.6          3.0           6.6          2.1  Iris-virginica
130           7.4          2.8           6.1          1.9  Iris-virginica
107           7.3          2.9           6.3          1.8  Iris-virginica
125           7.2          3.2           6.0          1.8  Iris-virginica
109           7.2          3.6           6.1          2.5  Iris-virginica

=== `irises.sort_values(by="sepal length", ascending=False).loc[5:10]`: Returns the observations between the one w
hose row ID is 5 and the one that is 10, in order of sepal-length, 5 and 10 are inclusive ===

    sepal length  sepal width  petal length  petal width      species
5            5.4          3.9           1.7          0.4  Iris-setosa
10           5.4          3.7           1.5          0.2  Iris-setosa

=== `irises['x'] = 3.14`: Creates a new column (variable) named ‘x’ and sets all values in column = 3.14 ===

   sepal length  sepal width  petal length  petal width      species     x
0           5.1          3.5           1.4          0.2  Iris-setosa  3.14
1           4.9          3.0           1.4          0.2  Iris-setosa  3.14
2           4.7          3.2           1.3          0.2  Iris-setosa  3.14
3           4.6          3.1           1.5          0.2  Iris-setosa  3.14
4           5.0          3.6           1.4          0.2  Iris-setosa  3.14

=== irises.rename(columns={'species': 'type'}): Change the name of a column (variable) ===

     sepal length  sepal width  petal length  petal width            type  \
0             5.1          3.5           1.4          0.2     Iris-setosa
1             4.9          3.0           1.4          0.2     Iris-setosa
2             4.7          3.2           1.3          0.2     Iris-setosa
3             4.6          3.1           1.5          0.2     Iris-setosa
4             5.0          3.6           1.4          0.2     Iris-setosa
5             5.4          3.9           1.7          0.4     Iris-setosa
6             4.6          3.4           1.4          0.3     Iris-setosa
7             5.0          3.4           1.5          0.2     Iris-setosa
8             4.4          2.9           1.4          0.2     Iris-setosa
9             4.9          3.1           1.5          0.1     Iris-setosa
10            5.4          3.7           1.5          0.2     Iris-setosa
11            4.8          3.4           1.6          0.2     Iris-setosa
12            4.8          3.0           1.4          0.1     Iris-setosa
13            4.3          3.0           1.1          0.1     Iris-setosa
14            5.8          4.0           1.2          0.2     Iris-setosa
15            5.7          4.4           1.5          0.4     Iris-setosa
16            5.4          3.9           1.3          0.4     Iris-setosa
17            5.1          3.5           1.4          0.3     Iris-setosa
18            5.7          3.8           1.7          0.3     Iris-setosa
19            5.1          3.8           1.5          0.3     Iris-setosa
20            5.4          3.4           1.7          0.2     Iris-setosa
21            5.1          3.7           1.5          0.4     Iris-setosa
22            4.6          3.6           1.0          0.2     Iris-setosa
23            5.1          3.3           1.7          0.5     Iris-setosa
24            4.8          3.4           1.9          0.2     Iris-setosa
25            5.0          3.0           1.6          0.2     Iris-setosa
26            5.0          3.4           1.6          0.4     Iris-setosa
27            5.2          3.5           1.5          0.2     Iris-setosa
28            5.2          3.4           1.4          0.2     Iris-setosa
29            4.7          3.2           1.6          0.2     Iris-setosa
..            ...          ...           ...          ...             ...
120           6.9          3.2           5.7          2.3  Iris-virginica
121           5.6          2.8           4.9          2.0  Iris-virginica
122           7.7          2.8           6.7          2.0  Iris-virginica
123           6.3          2.7           4.9          1.8  Iris-virginica
124           6.7          3.3           5.7          2.1  Iris-virginica
125           7.2          3.2           6.0          1.8  Iris-virginica
126           6.2          2.8           4.8          1.8  Iris-virginica
127           6.1          3.0           4.9          1.8  Iris-virginica
128           6.4          2.8           5.6          2.1  Iris-virginica
129           7.2          3.0           5.8          1.6  Iris-virginica
130           7.4          2.8           6.1          1.9  Iris-virginica
131           7.9          3.8           6.4          2.0  Iris-virginica
132           6.4          2.8           5.6          2.2  Iris-virginica
133           6.3          2.8           5.1          1.5  Iris-virginica
134           6.1          2.6           5.6          1.4  Iris-virginica
135           7.7          3.0           6.1          2.3  Iris-virginica
136           6.3          3.4           5.6          2.4  Iris-virginica
137           6.4          3.1           5.5          1.8  Iris-virginica
138           6.0          3.0           4.8          1.8  Iris-virginica
139           6.9          3.1           5.4          2.1  Iris-virginica
140           6.7          3.1           5.6          2.4  Iris-virginica
141           6.9          3.1           5.1          2.3  Iris-virginica
142           5.8          2.7           5.1          1.9  Iris-virginica
143           6.8          3.2           5.9          2.3  Iris-virginica
144           6.7          3.3           5.7          2.5  Iris-virginica
145           6.7          3.0           5.2          2.3  Iris-virginica
146           6.3          2.5           5.0          1.9  Iris-virginica
147           6.5          3.0           5.2          2.0  Iris-virginica
148           6.2          3.4           5.4          2.3  Iris-virginica
149           5.9          3.0           5.1          1.8  Iris-virginica

        x
```

```
0    3.14
1    3.14
2    3.14
3    3.14
4    3.14
5    3.14
6    3.14
7    3.14
8    3.14
9    3.14
10   3.14
11   3.14
12   3.14
13   3.14
14   3.14
15   3.14
16   3.14
17   3.14
18   3.14
19   3.14
20   3.14
21   3.14
22   3.14
23   3.14
24   3.14
25   3.14
26   3.14
27   3.14
28   3.14
29   3.14
..    ...
120  3.14
121  3.14
122  3.14
123  3.14
124  3.14
125  3.14
126  3.14
127  3.14
128  3.14
129  3.14
130  3.14
131  3.14
132  3.14
133  3.14
134  3.14
135  3.14
136  3.14
137  3.14
138  3.14
139  3.14
140  3.14
141  3.14
142  3.14
143  3.14
144  3.14
145  3.14
146  3.14
147  3.14
148  3.14
149  3.14

[150 rows x 6 columns]

=== `del irises['x']`: Removes a column ===

   sepal length  sepal width  petal length  petal width      species
0           5.1          3.5           1.4          0.2  Iris-setosa
1           4.9          3.0           1.4          0.2  Iris-setosa
2           4.7          3.2           1.3          0.2  Iris-setosa
3           4.6          3.1           1.5          0.2  Iris-setosa
4           5.0          3.6           1.4          0.2  Iris-setosa
```

## Merging data frames: join operations

Another useful operation on data frames is merging (http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.merge.html).

For instance, consider the following two tables, A and B:

| country | year | cases |
|---|---|---|
| Afghanistan | 1999 | 745 |
| Brazil | 1999 | 37737 |
| China | 1999 | 212258 |

| | | |
|---|---|---|
| Afghanistan | 2000 | 2666 |
| Brazil | 2000 | 80488 |
| China | 2000 | 213766 |

| country | year | population |
|---|---|---|
| Afghanistan | 1999 | 19987071 |
| Brazil | 1999 | 172006362 |
| China | 1999 | 1272915272 |
| Afghanistan | 2000 | 20595360 |
| Brazil | 2000 | 174504898 |
| China | 2000 | 1280428583 |

Suppose we wish to combine these into a single table, `C`:

| country | year | cases | population |
|---|---|---|---|
| Afghanistan | 1999 | 745 | 19987071 |
| Brazil | 1999 | 37737 | 172006362 |
| China | 1999 | 212258 | 1272915272 |
| Afghanistan | 2000 | 2666 | 20595360 |
| Brazil | 2000 | 80488 | 174504898 |
| China | 2000 | 213766 | 1280428583 |

In Pandas, you can perform this merge using the [`.merge()` function (http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.merge.html)](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.merge.html):

```
C = A.merge (B, on=['country', 'year'])
```

In this call, the `on=` parameter specifies the list of column names to use to align or "match" the two tables, `A` and `B`. By default, `merge()` will only include rows from `A` and `B` where all keys match between the two tables.

The following code cell demonstrates this functionality.

In [5]:
```
A_csv = """country,year,cases
Afghanistan,1999,745
Brazil,1999,37737
China,1999,212258
Afghanistan,2000,2666
Brazil,2000,80488
China,2000,213766"""

with StringIO(A_csv) as fp:
    A = pd.read_csv(fp)
print("=== A ===")
display(A)
```

```
=== A ===
```

| | country | year | cases |
|---|---|---|---|
| **0** | Afghanistan | 1999 | 745 |
| **1** | Brazil | 1999 | 37737 |
| **2** | China | 1999 | 212258 |
| **3** | Afghanistan | 2000 | 2666 |
| **4** | Brazil | 2000 | 80488 |
| **5** | China | 2000 | 213766 |

In [6]:
```
B_csv = """country,year,population
Afghanistan,1999,19987071
Brazil,1999,172006362
China,1999,1272915272
Afghanistan,2000,20595360
Brazil,2000,174504898
China,2000,1280428583"""

with StringIO(B_csv) as fp:
    B = pd.read_csv(fp)
print("\n=== B ===")
display(B)
```

```
=== B ===
```

| | country | year | population |
|---|---|---|---|

| | | | |
|---|---|---|---|
| **0** | Afghanistan | 1999 | 19987071 |
| **1** | Brazil | 1999 | 172006362 |
| **2** | China | 1999 | 1272915272 |
| **3** | Afghanistan | 2000 | 20595360 |
| **4** | Brazil | 2000 | 174504898 |
| **5** | China | 2000 | 1280428583 |

In [7]:
```
C = A.merge(B, on=['country', 'year'])
print("\n=== C = merge(A, B) ===")
display(C)
```

```
=== C = merge(A, B) ===
```

| | country | year | cases | population |
|---|---|---|---|---|
| **0** | Afghanistan | 1999 | 745 | 19987071 |
| **1** | Brazil | 1999 | 37737 | 172006362 |
| **2** | China | 1999 | 212258 | 1272915272 |
| **3** | Afghanistan | 2000 | 2666 | 20595360 |
| **4** | Brazil | 2000 | 80488 | 174504898 |
| **5** | China | 2000 | 213766 | 1280428583 |

**Joins.** This default behavior of keeping only rows that match both input frames is an example of what relational database systems call an *inner-join* operation. But there are several other types of joins.

- *Inner-join (A, B)* (default): Keep only rows of `A` and `B` where the on-keys match in both.
- *Outer-join (A, B)*: Keep all rows of both frames, but merge rows when the on-keys match. For non-matches, fill in missing values with not-a-number (`NaN`) values.
- *Left-join (A, B)*: Keep all rows of `A`. Only merge rows of `B` whose on-keys match `A`.
- *Right-join (A, B)*: Keep all rows of `B`. Only merge rows of `A` whose on-keys match `B`.

You can use `merge`'s `how=...` parameter, which takes the (string) values, `'inner'`, `'outer'`, `'left'`, and `'right'`. Here are some examples of these types of joins.

In [8]:
```
with StringIO("""x,y,z
bug,1,d
rug,2,d
lug,3,d
mug,4,d""") as fp:
    D = pd.read_csv(fp)
print("=== D ===")
display(D)

with StringIO("""x,y,w
hug,-1,e
smug,-2,e
rug,-3,e
tug,-4,e
bug,1,e""") as fp:
    E = pd.read_csv(fp)
print("\n=== E ===")
display(E)

print("\n=== Outer-join (D, E) ===")
display(D.merge(E, on=['x', 'y'], how='outer'))

print("\n=== Left-join (D, E) ===")
display(D.merge(E, on=['x', 'y'], how='left'))

print("\n=== Right-join (D, E) ===")
display(D.merge(E, on=['x', 'y'], how='right'))


print("\n=== Inner-join (D, E) ===")
display(D.merge(E, on=['x', 'y']))
```

```
=== D ===
```

| | x | y | z |
|---|---|---|---|
| **0** | bug | 1 | d |
| **1** | rug | 2 | d |
| **2** | lug | 3 | d |
| **3** | mug | 4 | d |

```
=== E ===
```

| | x | y | w |
|---|---|---|---|
| **0** | hug | -1 | e |
| **1** | smug | -2 | e |
| **2** | rug | -3 | e |
| **3** | tug | -4 | e |
| **4** | bug | 1 | e |

```
=== Outer-join (D, E) ===
```

| | x | y | z | w |
|---|---|---|---|---|
| **0** | bug | 1 | d | e |
| **1** | rug | 2 | d | NaN |
| **2** | lug | 3 | d | NaN |
| **3** | mug | 4 | d | NaN |
| **4** | hug | -1 | NaN | e |
| **5** | smug | -2 | NaN | e |
| **6** | rug | -3 | NaN | e |
| **7** | tug | -4 | NaN | e |

```
=== Left-join (D, E) ===
```

| | x | y | z | w |
|---|---|---|---|---|
| **0** | bug | 1 | d | e |
| **1** | rug | 2 | d | NaN |
| **2** | lug | 3 | d | NaN |
| **3** | mug | 4 | d | NaN |

```
=== Right-join (D, E) ===
```

| | x | y | z | w |
|---|---|---|---|---|
| **0** | bug | 1 | d | e |
| **1** | hug | -1 | NaN | e |
| **2** | smug | -2 | NaN | e |
| **3** | rug | -3 | NaN | e |
| **4** | tug | -4 | NaN | e |

```
=== Inner-join (D, E) ===
```

| | x | y | z | w |
|---|---|---|---|---|
| **0** | bug | 1 | d | e |

## Apply functions to data frames

Another useful primitive is `apply()`, which can apply a function to a data frame or to a series (column of the data frame).

For instance, suppose we wish to convert the year column in `C` into an abbrievated two-digit form. The following code will do it:

In [9]:
```
display(C)
```

| | country | year | cases | population |
|---|---|---|---|---|
| **0** | Afghanistan | 1999 | 745 | 19987071 |
| **1** | Brazil | 1999 | 37737 | 172006362 |
| **2** | China | 1999 | 212258 | 1272915272 |
| **3** | Afghanistan | 2000 | 2666 | 20595360 |
| **4** | Brazil | 2000 | 80488 | 174504898 |
| **5** | China | 2000 | 213766 | 1280428583 |

In [10]:
```
G = C.copy() # If you do not use copy function the original data frame is modified
G['year'] = G['year'].apply(lambda x: "'{:02d}".format(x % 100))
display(G)
```

| | country | year | cases | population |
|---|---|---|---|---|
| **0** | Afghanistan | '99 | 745 | 19987071 |
| **1** | Brazil | '99 | 37737 | 172006362 |
| **2** | China | '99 | 212258 | 1272915272 |
| **3** | Afghanistan | '00 | 2666 | 20595360 |
| **4** | Brazil | '00 | 80488 | 174504898 |
| **5** | China | '00 | 213766 | 1280428583 |

**Exercise 2** (2 points). Suppose you wish to compute the prevalence, which is the ratio of cases to the population.

The simplest way to do it is as follows:

```
G['prevalence'] = G['cases'] / G['population']
```

However, for this exercise, try to figure out how to use `apply()` to do it instead. To figure that out, you'll need to consult the documentation for `apply()` (http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.apply.html) or go online to find some hints.

Implement your solution in a function, `calc_prevalence(G)`, which given `G` returns a **new copy** `H` that has a column named `'prevalence'` holding the correctly computed prevalence values.

> **Note 0.** The emphasis on "new copy" is there to remind you that your function should *not* modify the input dataframe, `G`.
>
> **Note 1.** Although there is the easy solution above, the purpose of this exercise is to force you to learn more about how `apply()` works, so that you can "apply" it in more settings in the future.

In [11]:
```
def calc_prevalence(G):
    assert 'cases' in G.columns and 'population' in G.columns
    ### BEGIN SOLUTION
    def calc_ratio(observation):
        return observation['cases'] / observation['population']

    H = G.copy()
    H['prevalence'] = H.apply(calc_ratio, axis=1)
    return H
    ### END SOLUTION
```

In [12]:
```
# Test cell: `prevalence_test`

G_copy = G.copy()
H = calc_prevalence(G)
display(H) # Displayed `H` should have a 'prevalence' column

assert (G == G_copy).all().all(), "Did your function modify G? It shouldn't..."
assert set(H.columns) == (set(G.columns) | {'prevalence'}), "Check `H` again: it should have the same columns as `G
` plus a new column, `prevalence`."

Easy_prevalence_method = G['cases'] / G['population']
assert (H['prevalence'] == Easy_prevalence_method).all(), "One or more prevalence values is incorrect."

print("Prevalance values seem correct. But did you use `apply()?` Let's see...")

# Tests that you actually used `apply()` in your function:
def apply_fail():
    raise ValueError("Did you really use apply?")

setattr(pd.DataFrame, 'apply', apply_fail)
try:
    calc_prevalence(G)
except (ValueError, TypeError):
    print("You used `apply()`. You may have even used it as intended.")
else:
    assert False, "Are you sure you used `apply()`?"
finally:
    setattr(pd.DataFrame, 'apply', SAVE_APPLY)

print("\n(Passed!)")
```

| | country | year | cases | population | prevalence |
|---|---|---|---|---|---|
| **0** | Afghanistan | '99 | 745 | 19987071 | 0.000037 |
| **1** | Brazil | '99 | 37737 | 172006362 | 0.000219 |
| **2** | China | '99 | 212258 | 1272915272 | 0.000167 |

| | | | | | |
|---|---|---|---|---|---|
| **3** | Afghanistan | '00 | 2666 | 20595360 | 0.000129 |
| **4** | Brazil | '00 | 80488 | 174504898 | 0.000461 |
| **5** | China | '00 | 213766 | 1280428583 | 0.000167 |

```
Prevalance values seem correct. But did you use `apply()?` Let's see...
You used `apply()`. You may have even used it as intended.

(Passed!)
```

## Part 3 : Tibbles and Bits

Now let's start creating and manipulating tibbles.

In [13]:
```
import pandas as pd  # The suggested idiom
from io import StringIO

from IPython.display import display # For pretty-printing data frames
```

**Exercise 3** (3 points). Write a function, `canonicalize_tibble(X)`, that, given a tibble `X`, returns a new copy `Y` of `X` in *canonical order*. We say `Y` is in canonical order if it has the following properties.

1. The variables appear in sorted order by name, ascending from left to right.
2. The rows appear in lexicographically sorted order by variable, ascending from top to bottom.
3. The row labels (`Y.index`) go from 0 to `n-1`, where `n` is the number of observations.

For instance, here is a **non-canonical tibble** ...

| | c | a | b |
|---|---|---|---|
| 2 | hat | x | 1 |
| 0 | rat | y | 4 |
| 3 | cat | x | 2 |
| 1 | bat | x | 2 |

... and here is its **canonical counterpart.**

| | a | b | c |
|---|---|---|---|
| 0 | x | 1 | hat |
| 1 | x | 2 | bat |
| 2 | x | 2 | cat |
| 3 | y | 4 | rat |

A partial solution appears below, which ensures that Property 1 above holds. Complete the solution to ensure Properties 2 and 3 hold. Feel free to consult the [Pandas API (http://pandas.pydata.org/pandas-docs/stable/api.html)](http://pandas.pydata.org/pandas-docs/stable/api.html).

> **Hint**. For Property 3, you may find `reset_index()` handy: https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.reset_index.html (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.reset_index.html)

In [14]:
```
def canonicalize_tibble(X):
    # Enforce Property 1:
    var_names = sorted(X.columns)
    Y = X[var_names].copy()

    ### BEGIN SOLUTION
    # Enforce Property 2:
    Y.sort_values(by=var_names, inplace=True)

    # Enforce Property 3:
    Y.reset_index(drop=True, inplace=True)
    ### END SOLUTION
    return Y
```

In [15]:
```
# Test: `canonicalize_tibble_test`

# Test input
canonical_in_csv = """,c,a,b
2,hat,x,1
0,rat,y,4
3,cat,x,2
1,bat,x,2"""
```

```
with StringIO(canonical_in_csv) as fp:
    canonical_in = pd.read_csv(fp, index_col=0)
print("=== Input ===")
display(canonical_in)
print("")

# Test output solution
canonical_soln_csv = """,a,b,c
0,x,1,hat
1,x,2,bat
2,x,2,cat
3,y,4,rat"""

with StringIO(canonical_soln_csv) as fp:
    canonical_soln = pd.read_csv(fp, index_col=0)
print("=== True solution ===")
display(canonical_soln)
print("")

canonical_out = canonicalize_tibble(canonical_in)
print("=== Your computed solution ===")
display(canonical_out)
print("")

canonical_matches = (canonical_out == canonical_soln)
print("=== Matches? (Should be all True) ===")
display(canonical_matches)
assert canonical_matches.all().all()

print ("\n(Passed.)")
```

```
=== Input ===
```

| | c | a | b |
|---|---|---|---|
| **2** | hat | x | 1 |
| **0** | rat | y | 4 |
| **3** | cat | x | 2 |
| **1** | bat | x | 2 |

```
=== True solution ===
```

| | a | b | c |
|---|---|---|---|
| **0** | x | 1 | hat |
| **1** | x | 2 | bat |
| **2** | x | 2 | cat |
| **3** | y | 4 | rat |

```
=== Your computed solution ===
```

| | a | b | c |
|---|---|---|---|
| **0** | x | 1 | hat |
| **1** | x | 2 | bat |
| **2** | x | 2 | cat |
| **3** | y | 4 | rat |

```
=== Matches? (Should be all True) ===
```

| | a | b | c |
|---|---|---|---|
| **0** | True | True | True |
| **1** | True | True | True |
| **2** | True | True | True |
| **3** | True | True | True |

```
(Passed.)
```

**Exercise 4** (1 point). Write a function, `tibbles_are_equivalent(A, B)` to determine if two tibbles, `A` and `B`, are equivalent. "Equivalent" means that `A` and `B` have identical variables and observations, up to permutations. If `A` and `B` are equivalent, then the function should return `True`. Otherwise, it should return `False`.

The last condition, "up to permutations," means that the variables and observations might not appear in the table in the same order. For example, the following two tibbles are equivalent:

| a | b | c |
|---|---|---|
| x | 1 | hat |
| y | 2 | cat |
| z | 3 | bat |
| w | 4 | rat |

| b | c | a |
|---|---|---|
| 2 | cat | y |
| 3 | bat | z |
| 1 | hat | x |
| 4 | rat | w |

By contrast, the following table would not be equivalent to either of the above tibbles.

| a | b | c |
|---|---|---|
| 2 | y | cat |
| 3 | z | bat |
| 1 | x | hat |
| 4 | w | rat |

> **Note**: Unlike Pandas data frames, tibbles conceptually do not have row labels. So you should ignore row labels.

In [16]:

```
def tibbles_are_equivalent(A, B):
    """Given two tidy tables ('tibbles'), returns True iff they are
    equivalent.
    """
    ### BEGIN SOLUTION
    A_hat = canonicalize_tibble(A)
    B_hat = canonicalize_tibble(B)
    equal = (A_hat == B_hat)
    return equal.all().all()
    ### END SOLUTION
```

In [17]:

```
# Test: `tibble_are_equivalent_test`

A = pd.DataFrame(columns=['a', 'b', 'c'],
                 data=list(zip (['x', 'y', 'z', 'w'],
                                [1, 2, 3, 4],
                                ['hat', 'cat', 'bat', 'rat'])))
print("=== Tibble A ===")
display(A)

# Permute rows and columns, preserving equivalence
import random

obs_ind_orig = list(range(A.shape[0]))
var_names = list(A.columns)

obs_ind = obs_ind_orig.copy()
while obs_ind == obs_ind_orig:
    random.shuffle(obs_ind)

while var_names == list(A.columns):
    random.shuffle(var_names)

B = A[var_names].copy()
B = B.iloc[obs_ind]

print ("=== Tibble B == A ===")
display(B)

print ("=== Tibble C != A ===")
C = A.copy()
C.columns = var_names
display(C)

assert tibbles_are_equivalent(A, B)
assert not tibbles_are_equivalent(A, C)
assert not tibbles_are_equivalent(B, C)

print ("\n(Passed.)")
```

```
=== Tibble A ===
```

| | a | b | c |
|---|---|---|---|

| | | | |
|---|---|---|---|
| **0** | x | 1 | hat |
| **1** | y | 2 | cat |
| **2** | z | 3 | bat |
| **3** | w | 4 | rat |

```
=== Tibble B == A ===
```

| | c | a | b |
|---|---|---|---|
| **0** | hat | x | 1 |
| **2** | bat | z | 3 |
| **3** | rat | w | 4 |
| **1** | cat | y | 2 |

```
=== Tibble C != A ===
```

| | c | a | b |
|---|---|---|---|
| **0** | x | 1 | hat |
| **1** | y | 2 | cat |
| **2** | z | 3 | bat |
| **3** | w | 4 | rat |

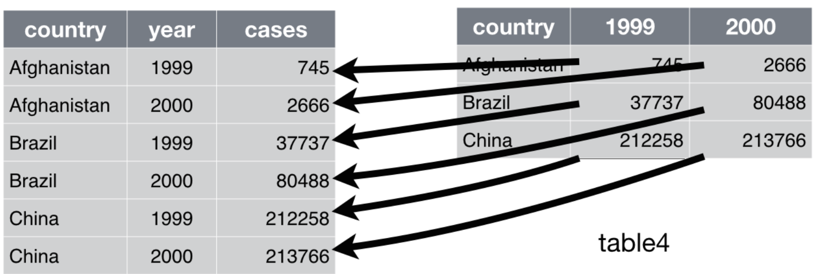
```
(Passed.)
```

# Basic tidying transformations: Melting and casting

Given a data set and a target set of variables, there are at least two common issues that require tidying.

## Melting

First, values often appear as columns. Table 4a is an example. To tidy up, you want to turn columns into rows:




Because this operation takes columns into rows, making a "fat" table more tall and skinny, it is sometimes called *melting*.

To melt the table, you need to do the following.

1. Extract the *column values* into a new variable. In this case, columns `"1999"` and `"2000"` of `table4` need to become the values of the variable, `"year"`.
2. Convert the values associated with the column values into a new variable as well. In this case, the values formerly in columns `"1999"` and `"2000"` become the values of the `"cases"` variable.

In the context of a melt, let's also refer to `"year"` as the new *key* variable and `"cases"` as the new *value* variable.

**Exercise 5** (4 points). Implement the melt operation as a function,

```
def melt(df, col_vals, key, value):
        ...
```

It should take the following arguments:

- `df`: the input data frame, e.g., `table4` in the example above;
- `col_vals`: a list of the column names that will serve as values; column `1999` & `2000` in example table
- `key`: name of the new variable, e.g., `year` in the example above;
- `value`: name of the column to hold the values, `cases` in the example above

- value: name of the column to hold the values: cases in the example above

> You may need to refer to the Pandas documentation to figure out how to create and manipulate tables. The bits related to indexing (http://pandas.pydata.org/pandas-docs/stable/indexing.html) and merging (http://pandas.pydata.org/pandas-docs/stable/merging.html) may be especially helpful.

In [18]:
```python
def melt(df, col_vals, key, value):
    assert type(df) is pd.DataFrame
    ### BEGIN SOLUTION
    keep_vars = df.columns.difference(col_vals)
    melted_sections = []
    for c in col_vals:
        melted_c = df[keep_vars].copy()
        melted_c[key] = c
        melted_c[value] = df[c]
        melted_sections.append(melted_c)
    melted = pd.concat(melted_sections)
    return melted
    ### END SOLUTION
```

In [19]:
```python
# Test: `melt_test`

table4a = pd.read_csv('table4a.csv')
print("\n=== table4a ===")
display(table4a)

m_4a = melt(table4a, col_vals=['1999', '2000'], key='year', value='cases')
print("=== melt(table4a) ===")
display(m_4a)

table4b = pd.read_csv('table4b.csv')
print("\n=== table4b ===")
display(table4b)

m_4b = melt(table4b, col_vals=['1999', '2000'], key='year', value='population')
print("=== melt(table4b) ===")
display(m_4b)

m_4 = pd.merge(m_4a, m_4b, on=['country', 'year'])
print ("\n=== inner-join(melt(table4a), melt (table4b)) ===")
display(m_4)

m_4['year'] = m_4['year'].apply (int)

table1 = pd.read_csv('table1.csv')
print ("=== table1 (target solution) ===")
display(table1)
assert tibbles_are_equivalent(table1, m_4)
print ("\n(Passed.)")
```

```
=== table4a ===
```

| | country | 1999 | 2000 |
|---|---|---|---|
| 0 | Afghanistan | 745 | 2666 |
| 1 | Brazil | 37737 | 80488 |
| 2 | China | 212258 | 213766 |

```
=== melt(table4a) ===
```

| | country | year | cases |
|---|---|---|---|
| 0 | Afghanistan | 1999 | 745 |
| 1 | Brazil | 1999 | 37737 |
| 2 | China | 1999 | 212258 |
| 0 | Afghanistan | 2000 | 2666 |
| 1 | Brazil | 2000 | 80488 |
| 2 | China | 2000 | 213766 |

```
=== table4b ===
```

| | country | 1999 | 2000 |
|---|---|---|---|
| 0 | Afghanistan | 19987071 | 20595360 |
| 1 | Brazil | 172006362 | 174504898 |
| 2 | China | 1272915272 | 1280428583 |

```
=== melt(table4b) ===
```

| | country | year | population |
|---|---|---|---|
| **0** | Afghanistan | 1999 | 19987071 |
| **1** | Brazil | 1999 | 172006362 |
| **2** | China | 1999 | 1272915272 |
| **0** | Afghanistan | 2000 | 20595360 |
| **1** | Brazil | 2000 | 174504898 |
| **2** | China | 2000 | 1280428583 |

```
=== inner-join(melt(table4a), melt (table4b)) ===
```

| | country | year | cases | population |
|---|---|---|---|---|
| **0** | Afghanistan | 1999 | 745 | 19987071 |
| **1** | Brazil | 1999 | 37737 | 172006362 |
| **2** | China | 1999 | 212258 | 1272915272 |
| **3** | Afghanistan | 2000 | 2666 | 20595360 |
| **4** | Brazil | 2000 | 80488 | 174504898 |
| **5** | China | 2000 | 213766 | 1280428583 |

```
=== table1 (target solution) ===
```

| | country | year | cases | population |
|---|---|---|---|---|
| **0** | Afghanistan | 1999 | 745 | 19987071 |
| **1** | Afghanistan | 2000 | 2666 | 20595360 |
| **2** | Brazil | 1999 | 37737 | 172006362 |
| **3** | Brazil | 2000 | 80488 | 174504898 |
| **4** | China | 1999 | 212258 | 1272915272 |
| **5** | China | 2000 | 213766 | 1280428583 |

```
(Passed.)
```

## Casting

The second most common issue is that an observation might be split across multiple rows. Table 2 is an example. To tidy up, you want to merge rows:

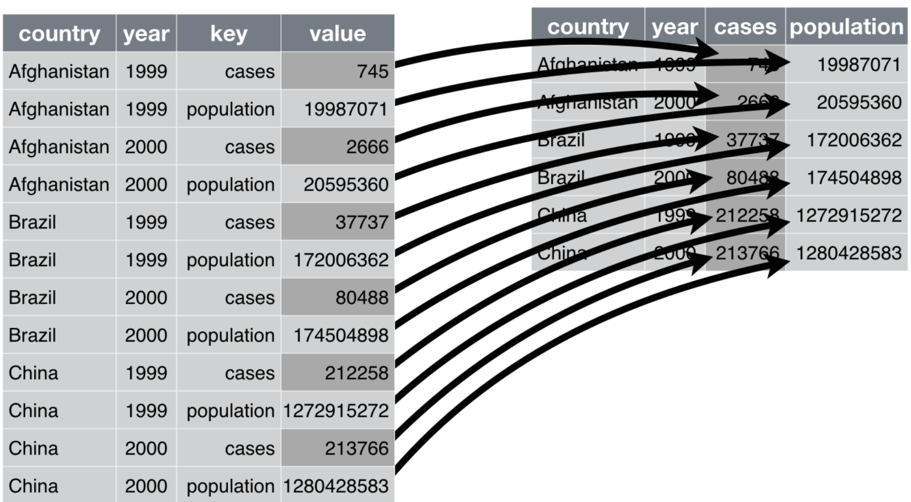



table2

Because this operation is the moral opposite of melting, and "rebuilds" observations from parts, it is sometimes called *casting*.

> Melting and casting are Wickham's terms from [his original paper on tidying data (http://www.jstatsoft.org/v59/i10/paper)]. In his more recent writing, [on which this tutorial is based (http://r4ds.had.co.nz/tidy-data.html)], he refers to the same operation as *gathering*. Again, this term comes from Wickham's original paper, whereas his more recent summaries use the term *spreading*.

The signature of a cast is similar to that of melt. However, you only need to know the `key`, which is column of the input table containing new variable names, and the `value`, which is the column containing corresponding values.

**Exercise 6** (4 points). Implement a function to cast a data frame into a tibble, given a key column containing new variable names and a value column containing the corresponding cells.

We've given you a partial solution that

- verifies that the given `key` and `value` columns are actual columns of the input data frame;
- computes the list of columns, `fixed_vars`, that should remain unchanged; and
- initializes and empty tibble.

Observe that we are asking your `cast()` to accept an optional parameter, `join_how`, that may take the values `'outer'` or `'inner'` (with `'outer'` as the default). Why do you need such a parameter?

In [20]:
```
def cast(df, key, value, join_how='outer'):
    """Casts the input data frame into a tibble,
    given the key column and value column.
    """
    assert type(df) is pd.DataFrame
    assert key in df.columns and value in df.columns
    assert join_how in ['outer', 'inner']

    fixed_vars = df.columns.difference([key, value])
    tibble = pd.DataFrame(columns=fixed_vars) # empty frame

    ### BEGIN SOLUTION
    new_vars = df[key].unique()
    for v in new_vars:
        df_v = df[df[key] == v]
        del df_v[key]
        df_v = df_v.rename(columns={value: v})
        tibble = tibble.merge(df_v,
                              on=list(fixed_vars),
                              how=join_how)
    ### END SOLUTION

    return tibble
```

In [21]:
```
# Test: `cast_test`

table2 = pd.read_csv('table2.csv')
print('=== table2 ===')
display(table2)

print('\n=== tibble2 = cast (table2, "type", "count") ===')
tibble2 = cast(table2, 'type', 'count')
display(tibble2)

assert tibbles_are_equivalent(table1, tibble2)
print('\n(Passed.)')
```

```
=== table2 ===
```

| | country | year | type | count |
|---|---|---|---|---|
| **0** | Afghanistan | 1999 | cases | 745 |
| **1** | Afghanistan | 1999 | population | 19987071 |
| **2** | Afghanistan | 2000 | cases | 2666 |
| **3** | Afghanistan | 2000 | population | 20595360 |
| **4** | Brazil | 1999 | cases | 37737 |
| **5** | Brazil | 1999 | population | 172006362 |
| **6** | Brazil | 2000 | cases | 80488 |
| **7** | Brazil | 2000 | population | 174504898 |
| **8** | China | 1999 | cases | 212258 |
| **9** | China | 1999 | population | 1272915272 |
| **10** | China | 2000 | cases | 213766 |
| **11** | China | 2000 | population | 1280428583 |

```
=== tibble2 = cast (table2, "type", "count") ===
```

| | country | year | cases | population |
|---|---|---|---|---|
| **0** | Afghanistan | 1999 | 745 | 19987071 |
| **1** | Afghanistan | 2000 | 2666 | 20595360 |
| **2** | Brazil | 1999 | 37737 | 172006362 |

| | | | | |
|---|---|---|---|---|
| 3 | Brazil | 2000 | 80488 | 174504898 |
| 4 | China | 1999 | 212258 | 1272915272 |
| 5 | China | 2000 | 213766 | 1280428583 |

```
(Passed.)
```

## Separating variables

Consider the following table.

In [22]:
```
table3 = pd.read_csv('table3.csv')
display(table3)
```

| | country | year | rate |
|---|---|---|---|
| 0 | Afghanistan | 1999 | 745/19987071 |
| 1 | Afghanistan | 2000 | 2666/20595360 |
| 2 | Brazil | 1999 | 37737/172006362 |
| 3 | Brazil | 2000 | 80488/174504898 |
| 4 | China | 1999 | 212258/1272915272 |
| 5 | China | 2000 | 213766/1280428583 |

In this table, the `rate` variable combines what had previously been the `cases` and `population` data. This example is an instance in which we might want to *separate* a column into two variables.

**Exercise 6** (3 points). Write a function that takes a data frame (`df`) and separates an existing column (`key`) into new variables (given by the list of new variable names, `into`).

How will the separation happen? The caller should provide a function, `splitter(x)`, that given a value returns a *list* containing the components. Observe that the partial solution below defines a default splitter, which uses the regular expression, `(\d+\.?\d+)`, to find all integer or floating-point values in a string input `x`.

In [23]:
```
import re

def default_splitter(text):
    """Searches the given spring for all integer and floating-point
    values, returning them as a list _of strings_.

    E.g., the call

      default_splitter('Give me $10.52 in exchange for 91 kitten stickers.')

    will return ['10.52', '91'].
    """
    fields = re.findall('(\d+\.?\d+)', text)
    return fields

def separate(df, key, into, splitter=default_splitter):
    """Given a data frame, separates one of its columns, the key,
    into new variables.
    """
    assert type(df) is pd.DataFrame
    assert key in df.columns

    # Hint: http://stackoverflow.com/questions/16236684/apply-pandas-function-to-column-to-create-multiple-new-colu
mns

    ### BEGIN SOLUTION
    def apply_splitter(text):
        fields = splitter(text)
        return pd.Series({into[i]: f for i, f in enumerate (fields)})

    fixed_vars = df.columns.difference([key])
    tibble = df[fixed_vars].copy()
    tibble_extra = df[key].apply(apply_splitter)
    return pd.concat([tibble, tibble_extra], axis=1)
    ### END SOLUTION
```

In [24]:
```
# Test: `separate_test`

print("=== Recall: table3 ===")
display(table3)

tibble3 = separate(table3, key='rate', into=['cases', 'population'])
print("\n=== tibble3 = separate (table3, ...) ===")
```

```
print("\n=== tibble3 = separate (table3, ...) ===")
display(tibble3)

assert 'cases' in tibble3.columns
assert 'population' in tibble3.columns
assert 'rate' not in tibble3.columns

tibble3['cases'] = tibble3['cases'].apply(int)
tibble3['population'] = tibble3['population'].apply(int)

assert tibbles_are_equivalent(tibble3, table1)
print("\n(Passed.)")
```

```
=== Recall: table3 ===
```

| | country | year | rate |
|---|---|---|---|
| **0** | Afghanistan | 1999 | 745/19987071 |
| **1** | Afghanistan | 2000 | 2666/20595360 |
| **2** | Brazil | 1999 | 37737/172006362 |
| **3** | Brazil | 2000 | 80488/174504898 |
| **4** | China | 1999 | 212258/1272915272 |
| **5** | China | 2000 | 213766/1280428583 |

```
=== tibble3 = separate (table3, ...) ===
```

| | country | year | cases | population |
|---|---|---|---|---|
| **0** | Afghanistan | 1999 | 745 | 19987071 |
| **1** | Afghanistan | 2000 | 2666 | 20595360 |
| **2** | Brazil | 1999 | 37737 | 172006362 |
| **3** | Brazil | 2000 | 80488 | 174504898 |
| **4** | China | 1999 | 212258 | 1272915272 |
| **5** | China | 2000 | 213766 | 1280428583 |

```
(Passed.)
```

**Exercise 7** (2 points). Implement the inverse of separate, which is `unite`. This function should take a data frame (`df`), the set of columns to combine (`cols`), the name of the new column (`new_var`), and a function that takes the subset of the `cols` variables from a single observation. It should return a new value for that observation.

In [25]:
```
def str_join_elements(x, sep=""):
    assert type(sep) is str
    return sep.join([str(xi) for xi in x])

def unite(df, cols, new_var, combine=str_join_elements):
    # Hint: http://stackoverflow.com/questions/13331698/how-to-apply-a-function-to-two-columns-of-pandas-dataframe
    ### BEGIN SOLUTION
    fixed_vars = df.columns.difference(cols)
    table = df[fixed_vars].copy()
    table[new_var] = df[cols].apply(combine, axis=1)
    return table
    ### END SOLUTION
```

In [26]:
```
# Test: `unite_test`

table3_again = unite(tibble3, ['cases', 'population'], 'rate',
                     combine=lambda x: str_join_elements(x, "/"))
display(table3_again)
assert tibbles_are_equivalent(table3, table3_again)

print("\n(Passed.)")
```

| | country | year | rate |
|---|---|---|---|
| **0** | Afghanistan | 1999 | 745/19987071 |
| **1** | Afghanistan | 2000 | 2666/20595360 |
| **2** | Brazil | 1999 | 37737/172006362 |
| **3** | Brazil | 2000 | 80488/174504898 |
| **4** | China | 1999 | 212258/1272915272 |
| **5** | China | 2000 | 213766/1280428583 |

```
(Passed.)
```

# Putting it all together

Let's use primitives to tidy up the original WHO TB data set. First, here is the raw data.

In [27]:
```
who_raw = pd.read_csv('who.csv')

print("=== WHO TB data set: {} rows x {} columns ===".format(who_raw.shape[0],
                                                         who_raw.shape[1]))
print("Column names:", who_raw.columns)

print("\n=== A few randomly selected rows ===")
import random
row_sample = sorted(random.sample(range(len(who_raw)), 5))
display(who_raw.iloc[row_sample])
```

```
=== WHO TB data set: 7240 rows x 60 columns ===
Column names: Index(['country', 'iso2', 'iso3', 'year', 'new_sp_m014', 'new_sp_m1524',
       'new_sp_m2534', 'new_sp_m3544', 'new_sp_m4554', 'new_sp_m5564',
       'new_sp_m65', 'new_sp_f014', 'new_sp_f1524', 'new_sp_f2534',
       'new_sp_f3544', 'new_sp_f4554', 'new_sp_f5564', 'new_sp_f65',
       'new_sn_m014', 'new_sn_m1524', 'new_sn_m2534', 'new_sn_m3544',
       'new_sn_m4554', 'new_sn_m5564', 'new_sn_m65', 'new_sn_f014',
       'new_sn_f1524', 'new_sn_f2534', 'new_sn_f3544', 'new_sn_f4554',
       'new_sn_f5564', 'new_sn_f65', 'new_ep_m014', 'new_ep_m1524',
       'new_ep_m2534', 'new_ep_m3544', 'new_ep_m4554', 'new_ep_m5564',
       'new_ep_m65', 'new_ep_f014', 'new_ep_f1524', 'new_ep_f2534',
       'new_ep_f3544', 'new_ep_f4554', 'new_ep_f5564', 'new_ep_f65',
       'newrel_m014', 'newrel_m1524', 'newrel_m2534', 'newrel_m3544',
       'newrel_m4554', 'newrel_m5564', 'newrel_m65', 'newrel_f014',
       'newrel_f1524', 'newrel_f2534', 'newrel_f3544', 'newrel_f4554',
       'newrel_f5564', 'newrel_f65'],
      dtype='object')

=== A few randomly selected rows ===
```

| | country | iso2 | iso3 | year | new_sp_m014 | new_sp_m1524 | new_sp_m2534 | new_sp_m3544 | new_sp_m4554 | new_sp_m5564 |
|---|---|---|---|---|---|---|---|---|---|---|
| **101** | Algeria | DZ | DZA | 2013 | NaN | NaN | NaN | NaN | NaN | NaN |
| **770** | Bermuda | BM | BMU | 2002 | NaN | NaN | NaN | NaN | NaN | NaN |
| **951** | Brazil | BR | BRA | 2009 | 328.0 | 4621.0 | 6399.0 | 5291.0 | 5058.0 | 2846.0 |
| **2108** | Egypt | EG | EGY | 2006 | 54.0 | 542.0 | 728.0 | 563.0 | 587.0 | 340.0 |
| **6280** | Thailand | TH | THA | 1984 | NaN | NaN | NaN | NaN | NaN | NaN |

5 rows × 60 columns

The data set has 7,240 rows and 60 columns. Here is how to decode the columns.

- Columns `'country'`, `'iso2'`, and `'iso3'` are different ways to designate the country and redundant, meaning you only really need to keep one of them.
- Column `'year'` is the year of the report and is a natural variable.
- Among columns `'new_sp_m014'` through `'newrel_f65'`, the `'new...'` prefix indicates that the column's values count new cases of TB. In this particular data set, all the data are for new cases.
- The short codes, `rel`, `ep`, `sn`, and `sp` describe the type of TB case. They stand for relapse, extrapulmonary, pulmonary not detectable by a pulmonary smear test ("smear negative"), and pulmonary detectable by such a test ("smear positive"), respectively.
- The codes `'m'` and `'f'` indicate the gender (male and female, respectively).
- The trailing numeric code indicates the age group: `014` is 0-14 years of age, `1524` for 15-24 years, `2534` for 25-34 years, etc., and `65` stands for 65 years or older.

In other words, it looks like you are likely to want to treat all the columns as values of multiple variables!

**Exercise 8** (3 points). As a first step, start with `who_raw` and create a new data frame, `who2`, with the following properties:

- All the `'new...'` columns of `who_raw` become values of a *single* variable, `case_type`. Store the counts associated with each `case_type` value as a new variable called `'count'`.
- Remove the `iso2` and `iso3` columns, since they are redundant with `country` (which you should keep!).
- Keep the `year` column as a variable.
- Remove all not-a-number (`NaN`) counts. *Hint*: You can test for a `NaN` using Python's [`math.isnan()` (https://docs.python.org/3/library/math.html)](https://docs.python.org/3/library/math.html).
- Convert the counts to integers. (Because of the presence of NaNs, the counts will be otherwise be treated as floating-point values, which is undesirable since you do not expect to see non-integer counts.)

In [28]:
```
from math import isnan

### BEGIN SOLUTION
# Melt value columns into a variable, `case_type`, associated with a new variable `count`:
col_vals = who_raw.columns.difference(['country', 'iso2', 'iso3', 'year'])
```

```
who2 = melt(who_raw, col_vals, 'case_type', 'count')

# Remove redundant iso2 and iso3 columns
del who2['iso2']
del who2['iso3']

# Remove NaNs
who2 = who2[who2['count'].apply(lambda x: not isnan(x))]

# Convert counts to ints
who2['count'] = who2['count'].apply(lambda x: int(x))

# Save this solution as "the" solution (master notebook only)
#who2.to_csv('who2_soln.csv', index=False)
### END SOLUTION
```

In [29]:

```
# Test: `who2_test`

print("=== First few rows of your solution ===")
display(who2.head())

print ("=== First few rows of the instructor's solution ===")
who2_soln = pd.read_csv('who2_soln.csv')
display(who2_soln.head())

# Check it
assert tibbles_are_equivalent(who2, who2_soln)
print ("\n(Passed.)")
```

```
=== First few rows of your solution ===
```

| | country | year | case_type | count |
|---|---|---|---|---|
| **60** | Albania | 2006 | new_ep_f014 | 7 |
| **61** | Albania | 2007 | new_ep_f014 | 1 |
| **62** | Albania | 2008 | new_ep_f014 | 3 |
| **63** | Albania | 2009 | new_ep_f014 | 2 |
| **64** | Albania | 2010 | new_ep_f014 | 2 |

```
=== First few rows of the instructor's solution ===
```

| | country | year | case_type | count |
|---|---|---|---|---|
| **0** | Albania | 2006 | new_ep_f014 | 7 |
| **1** | Albania | 2007 | new_ep_f014 | 1 |
| **2** | Albania | 2008 | new_ep_f014 | 3 |
| **3** | Albania | 2009 | new_ep_f014 | 2 |
| **4** | Albania | 2010 | new_ep_f014 | 2 |

```
(Passed.)
```

**Exercise 9** (5 points). Starting from your `who2` data frame, create a new tibble, `who3`, for which each `'key'` value is split into three new variables:

- `'type'`, to hold the TB type, having possible values of `rel`, `ep`, `sn`, and `sp`;
- `'gender'`, to hold the gender as a string having possible values of `female` and `male`; and
- `'age_group'`, to hold the age group as a string having possible values of `0-14`, `15-24`, `25-34`, `35-44`, `45-54`, `55-64`, and `65+`.

> The input data file is large enough that your solution might take a minute to run. But if it appears to be taking much more than that, you may want to revisit your approach.

In [30]:

```
### BEGIN SOLUTION
import re

def who_splitter(text):
    m = re.match("^new_?(rel|ep|sn|sp)_(f|m)(\\d{2,4})$", text)
    if m is None or len(m.groups()) != 3: # no match?
        return ['', '', '']

    fields = list(m.groups())
    if fields[1] == 'f':
        fields[1] = 'female'
    elif fields[1] == 'm':
        fields[1] = 'male'

    if fields[2] == '014':
        fields[2] = '0-14'
    elif fields[2] == '65':
```

```
        fields[2] = '65+'
    elif len(fields[2]) == 4 and fields[2].isdigit():
        fields[2] = fields[2][0:2] + '-' + fields[2][2:4]

    return fields

who3 = separate(who2,
                key='case_type',
                into=['type', 'gender', 'age_group'],
                splitter=who_splitter)

# Save this as the reference solution (master notebook only)
#who3.to_csv('who3_soln.csv', index=False)
### END SOLUTION"
```

In [31]:
```
# Test: `who3_test`

print("=== First few rows of your solution ===")
display(who3.head())

who3_soln = pd.read_csv('who3_soln.csv')
print("\n=== First few rows of the instructor's solution ===")
display(who3_soln.head())

assert tibbles_are_equivalent(who3, who3_soln)
print("\n(Passed.)")
```

```
=== First few rows of your solution ===
```

| | count | country | year | age_group | gender | type |
|---|---|---|---|---|---|---|
| **60** | 7 | Albania | 2006 | 0-14 | female | ep |
| **61** | 1 | Albania | 2007 | 0-14 | female | ep |
| **62** | 3 | Albania | 2008 | 0-14 | female | ep |
| **63** | 2 | Albania | 2009 | 0-14 | female | ep |
| **64** | 2 | Albania | 2010 | 0-14 | female | ep |

```
=== First few rows of the instructor's solution ===
```

| | count | country | year | age_group | gender | type |
|---|---|---|---|---|---|---|
| **0** | 7 | Albania | 2006 | 0-14 | female | ep |