DSS5201 Data Visualization

Week 5

Yuting Huang

NUS DSDS

2024-09-10

Tidy data

Artwork by Allison Horst

You have probably heard of the word “tidy” in your life.

- In terms of data analysis, **tidy data** refers to a standard way of mapping the meaning of data to its structure.
- Getting data into this format requires some upfront work, but it pays off in the long term.

Source: konmari.com

In this lecture, we will learn a consistent way to **organize** our data in Python using the principle known as **tidy data**.

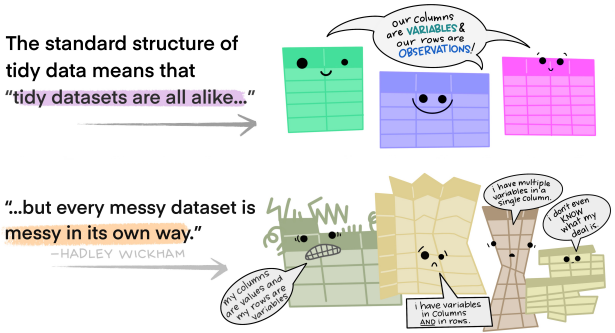
- Very helpful for lots of analysis and visualization tasks.

Getting our data into this format requires some work up front, but that work pays off in the long term.

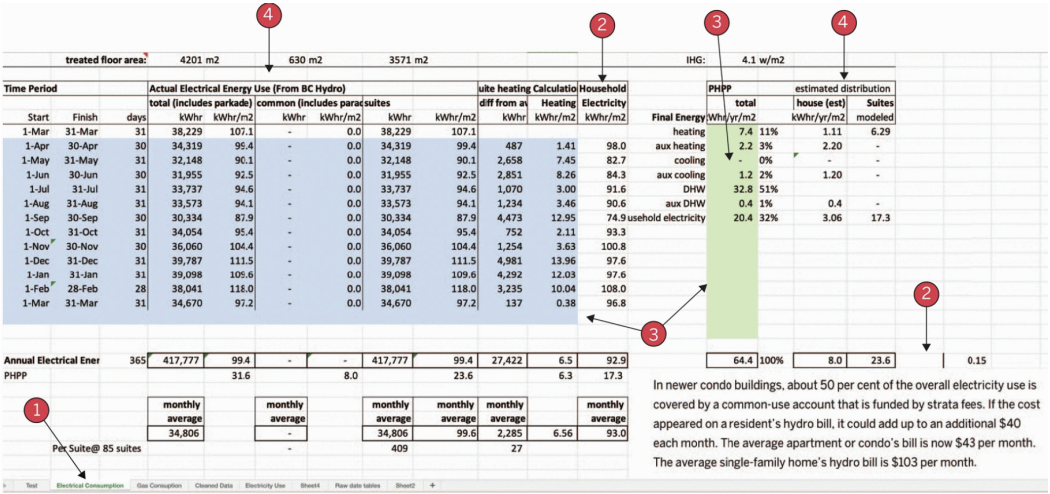
We will use the `pandas` data analysis package for tidy data.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import openpyxl
```

Most statistical data sets are **rectangular tables** made up of rows and columns.



Artwork by Allison Horst



treated floor area:	4201 m2	630 m2	3571 m2	IHG:	4.1 w/m2

Time Period			Actual Electrical Energy Use (From BC Hydro)						uite heating	Calculatio	Household
			total (includes parkade)		common (includes parac		suites		diff from av	Heating	Electricity
Start	Finish	days	kWhr	kWhr/m2	kWhr	kWhr/m2	kWhr	kWhr/m2	kWhr	kWhr/m2	kWhr/m2
1-Mar	31-Mar	31	38,229	107.1	-	0.0	38,229	107.1			
1-Apr	30-Apr	30	34,319	95.4	-	0.0	34,319	99.4	487	1.41	98.0
1-May	31-May	31	32,148	90.1	-	0.0	32,148	90.1	2,658	7.45	82.7
1-Jun	30-Jun	30	31,955	92.5	-	0.0	31,955	92.5	2,851	8.26	84.3
1-Jul	31-Jul	31	33,737	94.6	-	0.0	33,737	94.6	1,070	3.00	91.6
1-Aug	31-Aug	31	33,573	94.1	-	0.0	33,573	94.1	1,234	3.46	90.6
1-Sep	30-Sep	30	30,334	87.9	-	0.0	30,334	87.9	4,473	12.95	74.9
1-Oct	31-Oct	31	34,054	95.4	-	0.0	34,054	95.4	752	2.11	93.3
1-Nov	30-Nov	30	36,060	104.4	-	0.0	36,060	104.4	1,254	3.63	100.8
1-Dec	31-Dec	31	39,787	111.5	-	0.0	39,787	111.5	4,981	13.96	97.6
1-Jan	31-Jan	31	39,098	105.6	-	0.0	39,098	109.6	4,292	12.03	97.6
1-Feb	28-Feb	28	38,041	118.0	-	0.0	38,041	118.0	3,235	10.04	108.0
1-Mar	31-Mar	31	34,670	97.2	-	0.0	34,670	97.2	137	0.38	96.8
Annual Electrical Ener		365	417,777	99.4	-	-	417,777	99.4	27,422	6.5	92.9
PHPP				31.6		8.0		23.6		6.3	17.3
			monthly average		monthly average		monthly average	monthly average	monthly average		monthly average
			34,806		-		34,806	99.6	2,285	6.56	93.0
Per Suite@ 85 suites					-		409		27		

	PHPP		estimated distribution	
	total		house (est)	Suites
Final Energy	kWh/yr/m2		kWhr/yr/m2	modeled
heating	7.4	11%	1.11	6.29
aux heating	2.2	3%	2.20	-
cooling	-	0%	-	-
aux cooling	1.2	2%	1.20	-
DHW	32.8	51%		
aux DHW	0.4	1%	0.4	-
usehold electricity	20.4	32%	3.06	17.3
	64.4	100%	8.0	23.6

0.15

In newer condo buildings, about 50 per cent of the overall electricity use is covered by a common-use account that is funded by strata fees. If the cost appeared on a resident's hydro bill, it could add up to an additional $40 each month. The average apartment or condo's bill is now $43 per month. The average single-family home's hydro bill is $103 per month.

Source: Bartram et al (2022).

Data can be presented in different ways.

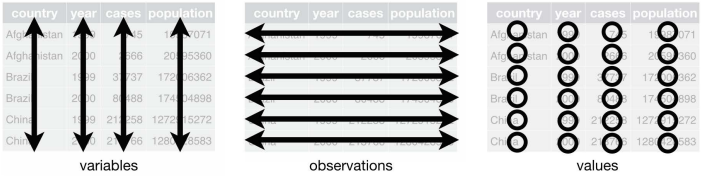
Name	Treatment A	Treatment B
John Smith	-	2
Jane Doe	16	11
Mary Johnson	3	1

Treatment	John Smith	Jane Doe	Mary Johnson
A	-	16	3
B	2	11	1

These are representations of the same underlying data, but they are **not** equally easy to use.

A data set is a collection of values.

1. Values are organized in two ways: every value belongs to a variable (column) and an observation (row).
2. A variable contains all values that measure the same underlying attribute across observations.
3. An observation contains all values measured across attributes.



Treatment	John Smith	Jane Doe	Mary Johnson
A	-	16	3
B	2	11	1

In this table, there are three variables:

- person's name, with three possible values
- treatment, with two possible values A or B.
- result, with six possible values, one of which is missing

There are six values.

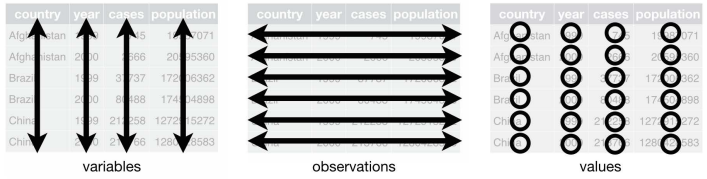
Each value is identified by the combination of person and treatment values.

This table will be much easier to work with because it is **tidy**.

id	Name	Treatment	Result
1	John Smith	A	-
2	Jane Doe	A	16
3	Mary Johnson	A	3
4	John Smith	B	2
5	Jane Doe	B	11
6	Mary Johnson	B	1

Tidy data is a standard way of structuring a data. There are three interrelated rules:

- Each variable forms a column.
- Each observation forms a row.
- Each cell contains a single value.



Take a look at the data on Stock prices. Are these tidy data?

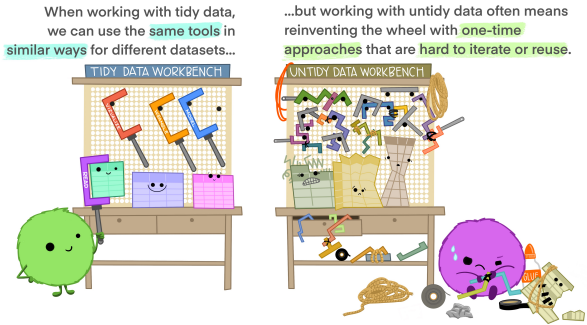
Date	Amazon	Boeing	Google
2009-01-01	174.90	173.55	174.34
2009-01-02	171.42	172.61	170.04

- Although the data are organized in a rectangular spreadsheet, they **do not** follow the definition for tidy format.

In a tidy data set, each variable should have its own column, and each observation should form its own row.

Date	Stock	Prices
2009-01-01	Amazon	174.90
2009-01-01	Boeing	173.55
2009-01-01	Google	174.30
2009-01-02	Amazon	171.42
2009-01-02	Boeing	172.61
2009-01-02	Google	170.04

- Placing variables in column allows the vectorized nature of `pandas` functions to take precedence.
- Most visualization functions are designed for tidy data. Having a consistent data structure means that we don't have to re-invent the tools to work with data.



Artwork by Allison Horst

The following data frames contain TB cases for countries in 1999 and 2000.

```
table1 = pd.read_excel("../data/wk5_tables.xlsx", sheet_name = "table1")
table1.head(2)
```

	country	year	cases	population
0	Afghanistan	1999	745	19987071
1	Afghanistan	2000	2666	20595360

```
table2 = pd.read_excel("../data/wk5_tables.xlsx", sheet_name = "table2")
table2.head(4)
```

	country	year	type	count
0	Afghanistan	1999	cases	745
1	Afghanistan	1999	population	19987071
2	Afghanistan	2000	cases	2666
3	Afghanistan	2000	population	20595360

Sketch out the process you would use for the following transformations:

1. Extract TB cases per country per year.
2. For each year, sort countries in ascending order of total cases.
3. For each country and year, compute the number of cases per 10,000 of population.

Which table is easier to work with? Why?

1. Extract TB cases per country per year.

```
table1.head(2)
```

```
       country  year  cases  population
0  Afghanistan  1999    745    19987071
1  Afghanistan  2000   2666    20595360
```

```
table2.query("type == 'cases'").head(2)
```

```
       country  year   type  count
0  Afghanistan  1999  cases    745
2  Afghanistan  2000  cases   2666
```

2. For each year, sort countries in ascending order of total cases.

```
table1.sort_values(["year", "cases"])
```

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

```
table2.query("type == 'cases'").sort_values(["year", "type"])
```

3. For each country and year, compute the number of cases per 10,000 of population.

```
table1_1 = table1.copy()
table1_1["per_capita"] = table1_1["cases"]/(table1_1["population"]/10000)
table1_1.head(2)
```

	country	year	cases	population	per_capita
0	Afghanistan	1999	745	19987071	0.372741
1	Afghanistan	2000	2666	20595360	1.294466

How to do this with `table2`?

- We need to first sort by `country`, `year`, and `type` to ensure the data are in the correct order.
- What's next?

```
table2_1 = table2.sort_values(["country", "year", "type"])
table2_1.head()
```

```
       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
```

- Group the data by `country` and `year`.
- The `lambda` function is applied to each group:
  - Take the first value in the `count` column, and divide it by the second value in the same column, adjusted by 10,000.

```
table2_1 = table2.groupby(["country", "year"]).apply(
    lambda x: (x["count"].values[0] / (x["count"].values[1]/10000)))
table2_1.head()
```

```
country      year
Afghanistan  1999    0.372741
             2000    1.294466
Brazil       1999    2.193930
             2000    4.612363
China        1999    1.667495
dtype: float64
```

- Lastly, reset the index and create a new column named `per_capita`.

```
table2_1 = table2_1.reset_index()
table2_1.columns = ["country", "year", "per_capita"]
table2_1.head()
```

```
       country  year  per_capita
0  Afghanistan  1999    0.372741
1  Afghanistan  2000    1.294466
2       Brazil  1999    2.193930
3       Brazil  2000    4.612363
4        China  1999    1.667495
```

Data can be untidy in many different ways.

These are the two most common ones:

- Column headers are values, instead of variable names.
  - We can solve this problem using `melt()`.
- Multiple variables are stored in one column.
  - Solve this with `pivot()`.

Most real analyses will require at least a little tidying.

Reshaping from wide to long

- Let's take a closer look at the TB cases data.

```
table4a = pd.read_excel("../data/wk5_tables.xlsx", sheet_name = "table4a")
table4a.head()
```

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

Notice that there are three variables:

- Country, with 3 values.
- Year, with 2 values.
- Number of TB cases, with 6 distinct values.

Not tidy. Problems to fix:

1. One of the variables, `year`, is stored in the column names.

   While this is okay for display, it is not tidy.

2. The number of TB cases is spread across columns too.

We need to `melt()` the columns into a new pair of variables. The information that the function requires are

1. The column(s) to use as identifier variables.
   - The **country** column.
2. The name of the column that will be created whose values form the column names for now.
   - This is the `var_name` argument. In this example we should call this variable **year**.
3. The name of the column that will be created whose values are currently residing in the cells of the data.
   - This is the `value_name` argument. In this example we should call this variable **cases**.

```
table4a_tidy = table4a.melt(id_vars = ["country"],
                            var_name = "year", value_name = "cases")
```

```
table4a_tidy = table4a.melt(id_vars = ["country"],
                            var_name = "year", value_name = "cases")
table4a_tidy.head()
```

	country	year	cases
0	Afghanistan	1999	745
1	Brazil	1999	37737
2	China	1999	212258
3	Afghanistan	2000	2666
4	Brazil	2000	80488

We can achieve the same result with other functions in pandas.

- The `stack()` function first sets `country` into row indices.
- … and turns the remaining columns (1999 and 2000) into rows.

```
table4a_tidy1 = table4a.set_index("country").stack().reset_index()
table4a_tidy1.columns = ["country", "year", "cases"]
table4a_tidy1.head()
```

```
       country  year   cases
0  Afghanistan  1999     745
1  Afghanistan  2000    2666
2       Brazil  1999   37737
3       Brazil  2000   80488
4        China  1999  212258
```

Reshaping from long to wide

Let's take a look at `table2` again – another representation of the same data.

- Each observation unit scatters across two rows.

```
table2 = pd.read_excel("../data/wk5_tables.xlsx", sheet_name = "table2")
table2.head(4)
```

	country	year	type	count
0	Afghanistan	1999	cases	745
1	Afghanistan	1999	population	19987071
2	Afghanistan	2000	cases	2666
3	Afghanistan	2000	population	20595360

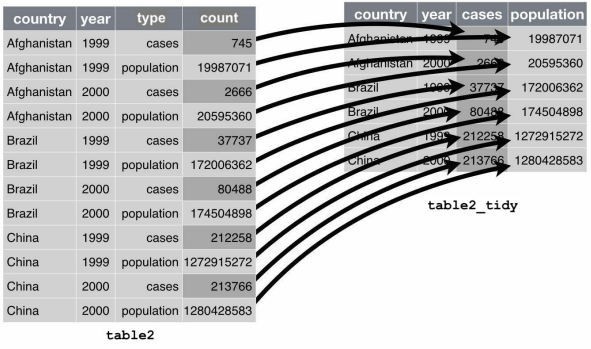
- The observation unit should be a country in each year.
- The `type` column contains two types of measurements for each ideal observation unit.

Wee need to `pivot()` the `type` and `count` columns into a single row for each observation.

1. The column that currently contains the unique identifier is **country**.
   - This is the `index` argument to the `pivot()` function.
2. The column that currently contains variable names is **type**.
   - This is the `columns` argument to `pivot()`.
3. The column that currently contains multiple values is **count**.
   - This is passed as the `values` argument to `pivot()`.

How to pivot()

```
table2_tidy = table2.pivot(index = ["country", "year"],
                           columns = "type", values = "count")
table2_tidy = table2_tidy.reset_index()
table2_tidy.columns.name = None
```



```
table2_tidy = table2.pivot(index = ["country", "year"],
                           columns = "type", values = "count")
table2_tidy
```

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

- After a `pivot()`, there are a couple of more steps we need to take to ensure the data frame is in the desired format.

1. Reset the row indices (country and year in this example).

```
# Reset row index
table2_tidy = table2_tidy.reset_index()
table2_tidy.head()
```

type	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

`reset_index()` converts the row indices (`country` and `year` in this example) to a default **integer index**.

- The original row indices are moved to column(s).

	country	year	type	count
0	Afghanistan	1999	cases	745
1	Afghanistan	1999	population	19987071
2	Afghanistan	2000	cases	2666
3	Afghanistan	2000	population	20595360

`pivot()` ↓

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

`reset_index()` →

	type	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	

2. Remove column index (type in this example).

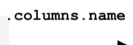
```
# Remove column index
table2_tidy.columns.name = None
table2_tidy.head()
```

	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

`.columns.name = None` removes the column index (`type` in this example).

- This further **simplifies** the data structure.

type	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



	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

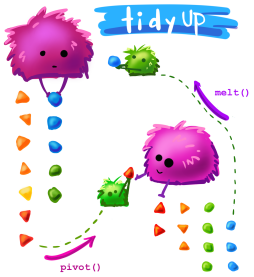
These two steps make the resulting data frame easier to work with.

An alternative is to use `pivot_table()`, which has similar syntax to `pivot()`.

- It allows for more flexible operations like aggregation (though not needed here).

```
table2_tidy1 = table2.pivot_table(
    index = ["country", "year"], columns = "type", values = "count")
table2_tidy1 = table2_tidy1.reset_index()
table2_tidy.columns.name = None
table2_tidy.head()
```

	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



- `melt()` makes a data frame tall and narrow.
- `pivot()` makes a data frame wide and short.

Not all long tables are tidy

A long table that is **not** tidy.

country	year	variable	value
Afghanistan	1999	cases	745
Afghanistan	1999	population	19987071
Afghanistan	2000	cases	2666
Afghanistan	2000	population	20595360
Brazil	1999	cases	37737
Brazil	1999	population	172006362
Brazil	2000	cases	80488
Brazil	2000	population	174504898

A long table that is tidy.

country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898

Not all wide tables are untidy

A wide table that is **not** tidy.

country	cases_1999	cases_2000	population_1999	population_2000
Afghanistan	745	2666	19987071	20595360
Brazil	37737	80488	172006362	174504898

A wide table that is tidy.

country	cases	population
Afghanistan	745	19987071
Brazil	37737	172006362

- Whether a data frame is tidy or not does not depend on whether it is long or wide.
- ... but on how the information in the data is organized, whether the structure aligns with tidy data principles.

The same TB data could also be stored like this.

```
table3 = pd.read_excel("../data/wk5_tables.xlsx", sheet_name = "table3")
table3.head(2)
```

```
       country  year           rate
0  Afghanistan  1999   745/19987071
1  Afghanistan  2000  2666/20595360
```

- Instead of TB counts and population, they are combined as a `rate` variable, seperated by a forward slash (/).
- We need to split them apart with `str.split()`.

How to str.split()

```
split_columns = table3["rate"].str.split("/", expand = True)
table3["case"] = split_columns[0]
table3["population"] = split_columns[1]
table3 = table3.drop(columns = "rate")
```

country	year	rate
Afghanistan	1999	**745** / 19987071
Afghanistan	2000	**2666** / 20595360
Brazil	1999	**37737** / 172006362
Brazil	2000	**80488** / 174504898
China	1999	**212258** / 1272915272
China	2000	**213766** / 1280428583

table3

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

It pulls apart a column into multiple columns, by splitting wherever a separate character appears.

- The argument `expand = True` expands the split strings into separate columns.

```
split_columns = table3["rate"].str.split("/", expand = True)
split_columns.head(2)
```

```
      0         1
0   745  19987071
1  2666  20595360
```

- Next, assemble case and population back into table3_tidy.

```
table3_tidy = table3.copy()
table3_tidy["case"] = split_columns[0]
table3_tidy["population"] = split_columns[1]
table3_tidy = table3_tidy.drop(columns = "rate")
table3_tidy
```

	country	year	case	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

We can use other methods in pandas to split the information in the rate column.

- str.partition() has a similar syntax as str.split().

```
table3_tidy1 = table3.copy()
partitions = table3_tidy1["rate"].str.partition("/")
partitions
```

```
        0  1           2
0     745  /    19987071
1    2666  /    20595360
2   37737  /   172006362
3   80488  /   174504898
4  212258  /  1272915272
5  213766  /  1280428583
```

- Next, assemble `case` and `population` back into `table3_tidy1`.

```
table3_tidy1["case"] = partitions[0]
table3_tidy1["population"] = partitions[2]
table3_tidy1 = table3_tidy1.drop(columns = "rate")
table3_tidy1
```

	country	year	case	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

- Alternatively, use `str.extract()` with a regular expression.
- The `r` in front of the string denotes a **raw string literal** in Python.
  - `(\d+)` denotes one or more digits.
  - `r"(\d+)/(\d+)"` captures the numbers before and after the forward slash `/`.

```
table3_tidy2 = table3.copy()
partitions = table3_tidy2["rate"].str.extract(r"(\d+)/(\d+)")
partitions.head(3)
```

```
       0          1
0    745   19987071
1   2666   20595360
2  37737  172006362
```

- Assemble the columns...

```
table3_tidy2["case"] = partitions[0]
table3_tidy2["population"] = partitions[1]
table3_tidy2 = table3_tidy2.drop(columns = "rate")
table3_tidy2
```

	country	year	case	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

Case study

We will use the Billboard Top 100 data that record the song rankings in 2000.

Let's first read in the data.

```
billboard = pd.read_csv("../data/wk5_billboard.csv")
billboard.head()
```

	artist	track	date.entered	...	wk74	wk75	wk
0	2 Pac	Baby Don't Cry (Keep...	2000-02-26	...	NaN	NaN	Na
1	2Ge+her	The Hardest Part Of ...	2000-09-02	...	NaN	NaN	Na
2	3 Doors Down	Kryptonite	2000-04-08	...	NaN	NaN	Na
3	3 Doors Down	Loser	2000-10-21	...	NaN	NaN	Na
4	504 Boyz	Wobble Wobble	2000-04-15	...	NaN	NaN	Na

[5 rows x 79 columns]

There are 317 songs entered the Billboard Top 100 in 2000.

- The first three columns of the data are variables describing the song:
  - `artist`, `track`, `date.entered`.
- The song will be included in this data set as long as it was in the top 100 at some point in 2000.
- Then we have 76 columns (`wk1` to `wk76`) that describe the rank of the song in each week after it entered.

This form of storage is **not tidy**, but it is useful for data entry.

The data are in **wide** format.

- To tidy it up, we will `melt()` it to make the data frame longer.

```
billboard_tidy = billboard.melt(
    id_vars = ["artist", "track", "date.entered"],
    var_name = "week", value_name = "rank")
billboard_tidy.head()
```

	artist	track	date.entered	week	rank
0	2 Pac	Baby Don't Cry (Keep...	2000-02-26	wk1	87.0
1	2Ge+her	The Hardest Part Of ...	2000-09-02	wk1	91.0
2	3 Doors Down	Kryptonite	2000-04-08	wk1	81.0
3	3 Doors Down	Loser	2000-10-21	wk1	76.0
4	504 Boyz	Wobble Wobble	2000-04-15	wk1	57.0

This is a **long** data frame.

- What happens if a song was in the Top 100 for less than 76 weeks?
- Take the first song in our data set as an example. The song was only in the Top 100 for a few weeks. All the remaining weeks are filled with NAs.

```
billboard_tidy.sort_values(by = ["artist", "track", "week"]).head(3)
```

```
     artist                    track date.entered  week  rank
0      2 Pac  Baby Don't Cry (Keep...   2000-02-26   wk1  87.0
2853   2 Pac  Baby Don't Cry (Keep...   2000-02-26  wk10   NaN
3170   2 Pac  Baby Don't Cry (Keep...   2000-02-26  wk11   NaN
```

We would like to sort each song by the week in which it made the Top 100.

- To sort them correctly, need to extract the week number first.

```
wk_int = billboard_tidy["week"].str.extract(r"(\d+)").astype(int)
billboard_tidy["week"] = wk_int
billboard_tidy.head()
```

	artist	track	date.entered	week	rank
0	2 Pac	Baby Don't Cry (Keep...	2000-02-26	1	87.0
1	2Ge+her	The Hardest Part Of ...	2000-09-02	1	91.0
2	3 Doors Down	Kryptonite	2000-04-08	1	81.0
3	3 Doors Down	Loser	2000-10-21	1	76.0
4	504 Boyz	Wobble Wobble	2000-04-15	1	57.0

- Now we are able to sort the data frame correctly.

```
billboard_tidy = billboard_tidy.sort_values(
    by = ["artist", "track", "week"])
billboard_tidy.head()
```

	artist	track	date.entered	week	rank
0	2 Pac	Baby Don't Cry (Keep...	2000-02-26	1	87.0
317	2 Pac	Baby Don't Cry (Keep...	2000-02-26	2	82.0
634	2 Pac	Baby Don't Cry (Keep...	2000-02-26	3	72.0
951	2 Pac	Baby Don't Cry (Keep...	2000-02-26	4	77.0
1268	2 Pac	Baby Don't Cry (Keep...	2000-02-26	5	87.0

Missing values (NaN) represent weeks in which the song wasn't in the list.

- We can safely removed these observations.

```
billboard_tidy = billboard_tidy.dropna(subset = ["rank"])
billboard_tidy.head(10)
```

	artist	track	date.entered	week	rank
0	2 Pac	Baby Don't Cry (Keep...	2000-02-26	1	87.0
317	2 Pac	Baby Don't Cry (Keep...	2000-02-26	2	82.0
634	2 Pac	Baby Don't Cry (Keep...	2000-02-26	3	72.0
951	2 Pac	Baby Don't Cry (Keep...	2000-02-26	4	77.0
1268	2 Pac	Baby Don't Cry (Keep...	2000-02-26	5	87.0
1585	2 Pac	Baby Don't Cry (Keep...	2000-02-26	6	94.0
1902	2 Pac	Baby Don't Cry (Keep...	2000-02-26	7	99.0
1	2Ge+her	The Hardest Part Of ...	2000-09-02	1	91.0
318	2Ge+her	The Hardest Part Of ...	2000-09-02	2	87.0
635	2Ge+her	The Hardest Part Of ...	2000-09-02	3	92.0

- We convert an untidy data frame with 317 rows and 79 columns to a tidy one with 5307 rows and 5 columns.
- Rows with missing rankings were removed.

```
billboard.shape
```

```
(317, 79)
```

```
billboard_tidy.shape
```

```
(5307, 5)
```

Notice that `date.entered` records the date a song track entered the ranking.

- This is not fully informative.
- We can use information in `date.entered` and `week` to recover the actual date that the song enters the Top 100.

```
df = billboard_tidy.sort_values(["artist", "track"]).copy()
df.head(3)
```

```
    artist                    track date.entered  week  rank
0     2 Pac  Baby Don't Cry (Keep...   2000-02-26     1  87.0
317   2 Pac  Baby Don't Cry (Keep...   2000-02-26     2  82.0
634   2 Pac  Baby Don't Cry (Keep...   2000-02-26     3  72.0
```

```
df["date.entered"] = pd.to_datetime(df["date.entered"])
df["date"] = df["date.entered"] + pd.to_timedelta(df["week"]-1, unit ="W")

np.random.seed(5201) # for reproducibility
df[["artist", "track", "date.entered", "rank", "date"]].sample(5)
```

	artist	track	date.entered	rank	date
2114	Moore, Mandy	I Wanna Be With You	2000-06-17	31.0	2000-07-29
6051	Barenaked Ladies	Pinch Me	2000-09-09	34.0	2001-01-20
1408	Jagged Edge	He Can't Love U	1999-12-11	15.0	2000-01-08
4052	Rimes, LeAnn	I Need You	2000-05-27	11.0	2000-08-19
6120	Evans, Sara	Born To Fly	2000-10-21	86.0	2001-03-03

We can use `billboard_tidy` to answer the following questions.

1. Find the Top 5 artists that have made the longest reign in the Top 100.

```
artists = billboard_tidy["artist"].value_counts()
artists = artists.reset_index(name = "count")
artists.head(5)
```

	artist	count
0	Creed	104
1	Lonestar	95
2	Destiny's Child	92
3	Sisqo	74
4	N'Sync	74

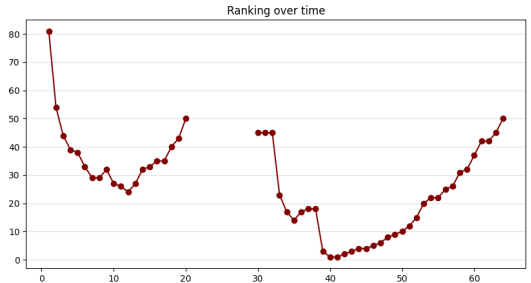
2. For each song track, find the number of weeks it stayed in the Top 100.

```
tracks = billboard_tidy[["artist", "track"]].value_counts()
tracks = tracks.reset_index(name = "count")
tracks.head()
```

	artist	track	count
0	Creed	Higher	57
1	Lonestar	Amazed	55
2	Hill, Faith	Breathe	53
3	3 Doors Down	Kryptonite	53
4	Creed	With Arms Wide Open	47

3. Visualize the ranking of *Lonestar's Amazed* over time.

```
df1 = billboard_tidy.query("artist == 'Lonestar' and track == 'Amazed'")
plt.figure(figsize=(10, 5))
plt.plot(df1["week"], df1["rank"], marker = "o", color = "maroon")
plt.title("Ranking over time")
plt.grid(True, axis = "y", alpha = 0.4)
plt.show()
```



Summary

The principles of tidy data may seem very obvious now. You might wonder if you will ever encounter a data set that is not tidy.

Unfortunately, however, most data that you will encounter will be untidy. There are two main reasons.

- Most people are not familiar with the principles of tidy data, and it is hard to derive them yourself unless you spent a lot of time working with data.
- Data are often organized to facilitate some use, other than analysis. For example, they can be organized to make data entry as easy as possible.

This means that for most real analyses, you will need to do some data tidying.

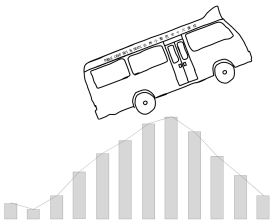
The first step of tidying data is always to figure out what are the variables and observations.

- Sometimes this is easy.
- Other times you will need to consult the people who originally generated the data.

The second step is to resolve one of two common problems:

- One variable spreads across multiple columns.
- One observation scatters across multiple rows.

Usually a data set will only suffer from one of these problems.



There are other `python` libraries that makes data manipulation simple and fast.

- `siuba` offers a user-friendly experience that mimics `R`'s `dplyr` syntax.
- `dfply` also mimics the functionality of `dplyr`.

If you come to Python from `R`, you will find these packages very easy to follow and use.