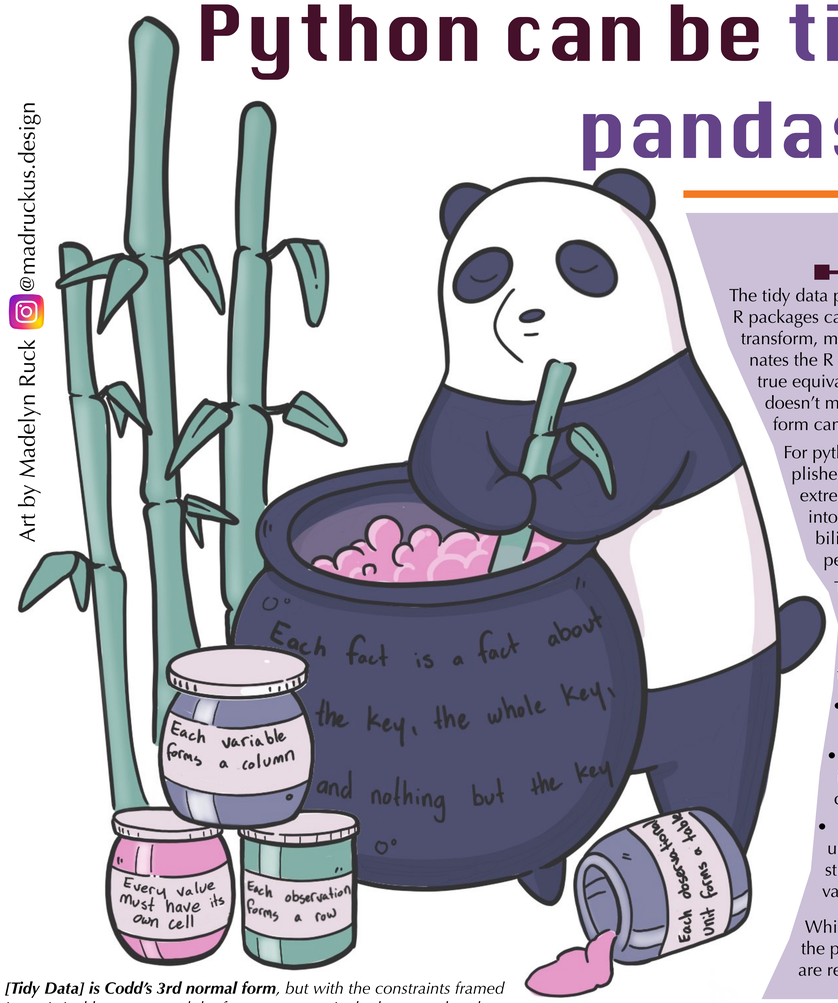
Python can be tidy too:
pandas recipes for normalizing data

Jenna Jordan

jennajordan.me · jenna-jordan · @JennaJrdn

School of Information Sciences
University of Illinois at Urbana-Champaign



Art by Madelyn Ruck @madruckus.design

> *[Tidy Data] is Codd's 3rd normal form*, but with the constraints framed in statistical language, and the focus put on a single dataset rather than the many connected datasets common in relational databases.
>
> - ***Hadley Wickham***, *Tidy Data (2014)*

Abstract

The tidy data philosophy is the foundation for the collection of R packages called the "tidyverse", which allow users to easily transform, model, and visualize data and currently dominates the R data science ecosystem. But while there is no true equivalent in the Python data science ecosystem, that doesn't mean the principles of tidy data and 3rd normal form cannot be applied in the Python ecosystem.

For python users, most data wrangling tasks are accomplished with the pandas library. The pandas library is extremely flexible and can manipulate tabular datasets into whatever form a user needs. However, this flexibility also means that it may be difficult for non-experts to fully utilize the library.

This cookbook was developed for those users who may be less familiar or comfortable with pandas, but want to transform their datasets into a tidy form within the Python ecosystem. The recipes in this cookbook will allow users to:

- **explore** datasets in order to detangle functional dependencies and identify candidate keys
- **transform** datasets by decomposing the table into new tables with proper functional dependencies and normalizing multivalued attributes
- **verify** that the new set of tables obey all necessary uniqueness constraints (primary keys), integrity constraints (foreign keys), and otherwise conform to a valid relational model

While actually creating the database is not necessary, the process of tidying the data will result in tables that are ready to be loaded into a relational database.

Example Datasets

My background is in political science - specifically Peace & Conflict studies - so I like using datasets from this domain. It helps that these datasets are most definitely not tidy! Each of the recipes will be demonstrated on datasets from either the Correlates of War Project (CoW) or the Uppsala Conflict Data Program (UCDP).

Get the full Tidy Pandas Cookbook on GitHub

https://github.com/jenna-jordan/tidy-pandas-cookbook

Explore

Identify Candidate Keys

A candidate key is some combination of attributes that can uniquely identify each observation in the dataset. One of these candidate keys will be chosen as the primary key - ideally the candidate key with the least number of attributes that makes the most sense given the actual meaning of the attribute.

```
ucdp_df.columns
```

```
Index(['conflict_id', 'location', 'side_a', 'side_a_id', 'side_a_2nd',
       'side_b', 'side_b_id', 'side_b_2nd', 'incompatibility',
       'territory_name', 'year', 'intensity_level', 'cumulative_intensity',
       'type_of_conflict', 'start_date', 'start_prec', 'start_date2',
       'start_prec2', 'ep_end', 'ep_end_date', 'ep_end_prec', 'gwno_a',
       'gwno_a_2nd', 'gwno_b', 'gwno_b_2nd', 'gwno_loc', 'region', 'version'],
      dtype='object')
```

```
find_candidate_keys(ucdp_df, max_columns=3)
```

```
[('conflict_id', 'year'),
 ('location', 'territory_name', 'year'),
 ('side_a', 'side_b', 'year'),
 ('side_a', 'side_b_id', 'year'),
 ('side_a_id', 'side_b', 'year'),
 ('side_a_id', 'side_b_id', 'year'),
 ('side_b', 'year', 'start_date'),
 ('side_b', 'year', 'start_date2'),
 ('side_b', 'year', 'gwno_a'),
 ('side_b_id', 'year', 'start_date'),
 ('side_b_id', 'year', 'start_date2'),
 ('side_b_id', 'year', 'gwno_a'),
 ('territory_name', 'year', 'start_date'),
 ('territory_name', 'year', 'gwno_loc')]
```

`find_candidate_keys()` is demonstrated on the UCDP dataset (`ucdp_df`). A list of all combinations of columns that can uniquely identify the row, with a maximum of 3 columns, are returned. The best candidate for primary key is the combination of `conflict_id` and `year`.

Find Dependent Columns

Identifying functional dependencies is the key to the normalization process. For a database to be in 3rd normal form, each attribute (column) must depend only on the table's primary key. So if there are a set of columns that can be functionally determined by another column (or set of columns), that is a hint that the dataset needs to be decomposed into two or more tables.

```
find_dependent_columns(ucdp_df, ['gwno_a'])
```

```
['side_a', 'side_a_id']
```

```
find_dependent_columns(ucdp_df, ['conflict_id', 'start_date2'])
```

```
['location',
 'side_a',
 'side_a_id',
 'incompatibility',
 'territory_name',
 'start_date',
 'start_prec',
 'start_prec2',
 'ep_end_date',
 'gwno_a',
 'gwno_b',
 'gwno_loc',
 'region']
```

You can check which columns can be functionally determined by a candidate key with `find_dependent_columns()`, demonstrated above on the UCDP dataset. `gwno_a` can functionally determine other columns that also describe countries on side A. The combination of `conflict_id` and the date the episode started (`start_date2`) can functionally determine a large subset of the columns in the `ucdp_df` dataset.

Transform

Multivalued Attributes

The ideal way to represent multivalued attributes, in accordance with 3rd normal form, is to isolate the attribute in its own table - one column (or set of columns) representing the entity, and one column for a single value. However, datasets found in the wild almost never conform to 3rd normal form.

	conflict_id	year	side_a	gwno_a	side_a_2nd	gwno_a_2nd
1643	13639	2016	Government of Niger	436	Government of Chad, Government of Nigeria	483, 475
2327	418	2012	Government of United States of America	2	Government of Afghanistan, Government of Franc...	700, 220, 663, 770
41	333	2015	Government of Afghanistan	700	Government of Pakistan, Government of United S...	770, 2
1247	261	1958	Government of Malaysia	820	Government of Australia, Government of New Zea...	900, 920, 200
1262	11347	2016	Government of Mali	432	Government of Armenia, Government of Austria, ...	371, 305, 771, 211, 434, 760, 439, 516, 811, 4...

```
split_lists(ucdp_df, pk_columns=['conflict_id', 'year'],
            list_column='gwno_a_2nd', delim=', ').head(10)
```

	conflict_id	year	gwno_a_2nd
0	13637	2015	770
1	13637	2015	2
2	13637	2016	770
3	13637	2016	2
4	13637	2017	2
5	13637	2018	2
6	333	1980	365
7	333	1981	365
8	333	1982	365
9	333	1983	365

When there are a wide variety of possible values, the multiple values may be crammed into one column and seperated by a delimiter (such as a comma). `split_lists()` is demonstrated on the UCDP dataset's `gwno_a_2nd` column, with the `conflict_id` and `year` columns as the unique identifier.

version4id	ccode	defense	neutrality	nonaggression	entente
1	200	1	0	1	0
1	235	1	0	1	0
2	200	0	0	0	1
2	380	0	0	0	1
3	240	1	0	1	1
3	240	1	0	1	1
3	245	1	0	1	1
3	245	1	0	1	1
3	255	1	0	1	1
3	255	1	0	1	1

```
de_dummify(cow_alliance_df, pk_columns=['version4id', 'ccode'],
           dummy_columns=['defense', 'neutrality',
           'nonaggression', 'entente'], col_name='trait')
```

	version4id	ccode	trait
0	1	200	defense
1	1	200	nonaggression
2	1	235	defense
3	1	235	nonaggression
4	2	200	entente
5	2	380	entente
6	3	240	defense
7	3	240	nonaggression
8	3	240	entente
9	3	240	defense

When there are only a few possibilities, the multiple values may be spread out over "dummy" columns with binary encoding. `de_dummify()` is demonstrated in the CoW Alliance dataset (`cow_alliance_df`) for the 4 alliance traits columns (`defense`, `neutrality`, `nonaggression`, and `entente`).

Decomposing tables

After identifying the functional dependencies, you will want to create new tables that conform to these functional dependencies. We have already determined which columns can be functionally determined by `conflict_id` and `start_date2` in `ucdp_df`. But some of these columns can be determined by `conflict_id` alone. The new table should have only those attributes that are functionally determined by the whole primary key. `decompose_table()` makes use of the `find_dependent_columns()` function to identify the proper columns and create a new dataframe that drops unnecessary columns and rows. However, further cleaning/investigation is still needed.

```
decompose_table(ucdp_df, ['conflict_id', 'start_date2'])
```

	conflict_id	start_date2	start_prec2	ep_end_date
0	13637	2015-03-03	1	NaN
1	333	1978-04-27	1	NaN
2	431	1979-12-27	1	1979-12-28
3	13692	2001-10-07	1	2001-11-13
4	215	1946-10-22	1	1946-12-31
...	...	...	...	...
806	402	1994-04-28	2	1994-07-04
807	318	1967-09-05	1	NaN
808	318	1967-09-05	1	1968-12-31
809	318	1973-04-04	1	NaN
810	318	1973-04-04	1	1979-12-21

811 rows × 4 columns

Verify

Referential Integrity

When you decompose a dataset into multiple smaller tables, you need a way to put them all back together again. In a database, foreign keys tell you which columns to merge on when combining tables - but they also ensure referential integrity. If a value exists in the foreign key column, it must also exist in the primary key column. For example, take these two tables:

```
pol.sample(5)
```

	id	name	startyear	endyear	type
701	3346	Zadar (Zara)	1816	1919	territory
1722	6829	Upper Yafai	1967	1990	territory
41	210	Netherlands	1816	1940	state
72	55	Grenada	1983	1985	territory
200	704	Uzbekistan	1991	2016	state

`pol` contains all states and territories in the CoW datasets, from 1816 - 2016.

```
tc[['number', 'gainer', 'entity', 'loser']]
```

	number	gainer	entity	loser
167	203	325	327	327
252	290	365	365	NaN
780	834	651	666	666
735	789	640	352	352
779	833	432	439	439

`tc` records all territorial changes, including the entity exchanged, the gainer, and the loser.

All of the polities in `tc`'s `gainer`, `entity`, and `loser` columns should also exist in `pol`'s `id` column, so we know which polity the identifier refers to.

`gainer` fails the referential integrity check.

```
check_ids_ref_integrity(primary_df = pol,
                        pdf_id = 'id',
                        related_df = tc,
                        rdf_fk = 'gainer')
```

```
False
```

From all three columns, there are three identifiers not present in the primary key.

```
check_ids_ref_integrity(primary_df = pol,
                        pdf_id = 'id',
                        related_df = tc,
                        rdf_fk = ['gainer', 'loser', 'entity'],
                        verbose = True)
```

```
{0, 1, 822}
```

By using the `verbose` flag, we can see which specific identifiers from the `gainer`, `loser`, and `entity` columns were not present in `pol`'s `id` column.

Uniqueness Constraint

In a database, the primary key must satisfy the uniqueness constraint - that is, the primary key value for one row cannot be repeated in another row. The primary key is the record's unique identifier, whether it is a single column or a combination of columns.

The table created by `decompose_table()`, to the left, visibly does not satisfy the uniqueness constraint. It's primary key, `conflict_id` and `start_date2`, is repeated for some rows. We can use `check_key_uniqueness()` to verify this.

```
check_key_uniqueness(ucdp_episode, ['conflict_id', 'start_date2'])
```

```
False
```

The issue is with `ep_end_date` - the original dataset was not tidy, and only recorded the episode end date for rows where the episode ended that year. For rows with a duplicated `conflict_id` and `start_date2`, we need to drop the row with a null `ep_end_date`. This is easily done by sorting on these rows, as the null dates will be placed last.

```
ucdp_episode = ucdp_episode.sort_values(by=['conflict_id', 'start_date2', 'ep_end_date']) \
                           .drop_duplicates(subset=['conflict_id', 'start_date2'], keep='first')
check_key_uniqueness(ucdp_episode, ['conflict_id', 'start_date2'])
```

```
True
```

The table now satisfies the uniqueness constraint, and is ready to be inserted into a database! Or, you can simply save this newly tidy dataframe to a CSV.