



Pythonic Data Cleaning With NumPy and Pandas

by Malay Agarwal 7 Comments [data-science](#) [intermediate](#)

Table of Contents

- [Dropping Columns in a DataFrame](#)
- [Changing the Index of a DataFrame](#)
- [Tidying up Fields in the Data](#)
- [Combining str Methods with NumPy to Clean Columns](#)
- [Cleaning the Entire Dataset Using the applymap Function](#)
- [Renaming Columns and Skipping Rows](#)
- [Python Data Cleaning: Recap and Resources](#)

Data scientists spend a large amount of their time cleaning datasets and getting them down to a form with which they can work. In fact, a lot of data scientists argue that the initial steps of obtaining and cleaning data constitute 80% of the job.

Therefore, if you are just stepping into this field or [planning to step into this field](#), it is important to be able to deal with messy data, whether that means missing values, inconsistent formatting, malformed records, or nonsensical outliers.

In this tutorial, we'll leverage Python's Pandas and NumPy libraries to clean data.

We'll cover the following:

- Dropping unnecessary columns in a DataFrame
- Changing the index of a DataFrame
- Using `.str()` methods to clean columns
- Using the `DataFrame.applymap()` function to clean the entire dataset, element-wise
- Renaming columns to a more recognizable set of labels
- Skipping unnecessary rows in a CSV file

Free Bonus: [Click here to get access to a free NumPy Resources Guide](#) that points you to the best tutorials, videos, and books for improving your NumPy skills.

Here are the datasets that we will be using:

- [BL-Flickr-Images-Book.csv](#) – A CSV file containing information about books from the British Library
- [university_towns.txt](#) – A text file containing names of college towns in every US state
- [olympics.csv](#) – A CSV file summarizing the participation of all countries in the Summer and Winter Olympics

You can download the datasets from Real Python's [GitHub repository](#) in order to follow the examples here.

Note: I recommend using Jupyter Notebooks to follow along.

This tutorial assumes a basic understanding of the Pandas and NumPy libraries, including Panda's workhorse [Series and DataFrame objects](#), common methods that can be applied to these objects, and familiarity with NumPy's [NaN values](#).

Let's import the required modules and get started!

```
Python
```

```
>>>
```

```
>>> import pandas as pd
>>> import numpy as np
```

Dropping Columns in a DataFrame

Often, you'll find that not all the categories of data in a dataset are useful to you. For example, you might have a dataset containing student information (name, grade, standard, parents' names, and address) but want to focus on analyzing student grades.

In this case, the address or parents' names categories are not important to you. Retaining these unneeded categories will take up unnecessary space and potentially also bog down runtime.

Pandas provides a handy way of removing unwanted columns or rows from a `DataFrame` with the `drop()` function. Let's look at a simple example where we drop a number of columns from a `DataFrame`.

First, let's create a `DataFrame` out of the CSV file 'BL-Flickr-Images-Book.csv'. In the examples below, we pass a relative path to `pd.read_csv`, meaning that all of the datasets are in a folder named `Datasets` in our current working directory:

```
Python
```

```
>>>
```

```

>>> df = pd.read_csv('Datasets/BL-Flickr-Images-Book.csv')
>>> df.head()

```

	Identifier	Edition Statement	Place of Publication
0	206	NaN	London
1	216	NaN	London; Virtue & Yorston
2	218	NaN	London
3	472	NaN	London
4	480	A new edition, revised, etc.	London

	Date of Publication	Publisher
0	1879 [1878]	S. Tinsley & Co.
1	1868	Virtue & Co.
2	1869	Bradbury, Evans & Co.
3	1851	James Darling
4	1857	Wertheim & Macintosh

	Title	Author
0	Walter Forbes. [A novel.] By A. A	A. A.
1	All for Greed. [A novel. The dedication signed...	A., A. A.
2	Love the Avenger. By the author of "All for Gr...	A., A. A.
3	Welsh Sketches, chiefly ecclesiastical, to the...	A., E. S.
4	[The World in which I live, and my place in it...	A., E. S.

	Contributors	Corporate Author
0	FORBES, Walter.	NaN
1	BLAZE DE BURY, Marie Pauline Rose - Baroness	NaN
2	BLAZE DE BURY, Marie Pauline Rose - Baroness	NaN
3	Appleyard, Ernest Silvanus.	NaN
4	BROOME, John Henry.	NaN

	Corporate Contributors	Former owner	Engraver	Issuance type
0	NaN	NaN	NaN	monographic
1	NaN	NaN	NaN	monographic
2	NaN	NaN	NaN	monographic
3	NaN	NaN	NaN	monographic
4	NaN	NaN	NaN	monographic

	Flickr URL
0	http://www.flickr.com/photos/britishlibrary/ta...
1	http://www.flickr.com/photos/britishlibrary/ta...
2	http://www.flickr.com/photos/britishlibrary/ta...
3	http://www.flickr.com/photos/britishlibrary/ta...
4	http://www.flickr.com/photos/britishlibrary/ta...

	Shelfmarks
0	British Library HMNTS 12641.b.30.
1	British Library HMNTS 12626.cc.2.
2	British Library HMNTS 12625.dd.1.
3	British Library HMNTS 10369.bbb.15.
4	British Library HMNTS 9007.d.28.

When we look at the first five entries using the `head()` method, we can see that a handful of columns provide ancillary information that would be helpful to the library but isn't very descriptive of the books themselves: `Edition Statement`, `Corporate Author`, `Corporate Contributors`, `Former owner`, `Engraver`, `Issuance type` and `Shelfmarks`.

We can drop these columns in the following way:

Python

>>>

```

>>> to_drop = ['Edition Statement',
...             'Corporate Author',
...             'Corporate Contributors',
...             'Former owner',
...             'Engraver',
...             'Contributors',
...             'Issuance type',
...             'Shelfmarks']

>>> df.drop(to_drop, inplace=True, axis=1)

```

Above, we defined a list that contains the names of all the columns we want to drop. Next, we call the `drop()` function on our object, passing in the `inplace` parameter as `True` and the `axis` parameter as `1`. This tells Pandas that we want the changes to be made directly in our object and that it should look for the values to be dropped in the columns of the object.

When we inspect the `DataFrame` again, we'll see that the unwanted columns have been removed:

```

Python >>>
>>> df.head()
  Identifier  Place of Publication Date of Publication \
0          206                London          1879 [1878]
1          216  London; Virtue & Yorston          1868
2          218                London          1869
3          472                London          1851
4          480                London          1857

  Publisher  Title \
0  S. Tinsley & Co.  Walter Forbes. [A novel.] By A. A
1  Virtue & Co.  All for Greed. [A novel. The dedication signed...
2  Bradbury, Evans & Co.  Love the Avenger. By the author of "All for Gr...
3  James Darling  Welsh Sketches, chiefly ecclesiastical, to the...
4  Wertheim & Macintosh  [The World in which I live, and my place in it...

  Author  Flickr URL
0  A. A.  http://www.flickr.com/photos/britishlibrary/ta...
1  A., A. A.  http://www.flickr.com/photos/britishlibrary/ta...
2  A., A. A.  http://www.flickr.com/photos/britishlibrary/ta...
3  A., E. S.  http://www.flickr.com/photos/britishlibrary/ta...
4  A., E. S.  http://www.flickr.com/photos/britishlibrary/ta...

```

Alternatively, we could also remove the columns by passing them to the `columns` parameter directly instead of separately specifying the labels to be removed and the axis where Pandas should look for the labels:

```

Python >>>
>>> df.drop(columns=to_drop, inplace=True)

```

This syntax is more intuitive and readable. What we're trying to do here is directly apparent.

If you know in advance which columns you'd like to retain, another option is to pass them to the `usecols` argument of `pd.read_csv`.

Changing the Index of a DataFrame

A Pandas `Index` extends the functionality of NumPy arrays to allow for more versatile slicing and labeling. In many cases,

it is helpful to use a uniquely valued identifying field of the data as its index.

For example, in the dataset used in the previous section, it can be expected that when a librarian searches for a record, they may input the unique identifier (values in the `Identifier` column) for a book:

```
Python >>>
>>> df['Identifier'].is_unique
True
```

Let's replace the existing index with this column using `set_index`:

```
Python >>>
>>> df = df.set_index('Identifier')
>>> df.head()
```

	Place of Publication	Date of Publication	\
206	London	1879 [1878]	
216	London; Virtue & Yorston	1868	
218	London	1869	
472	London	1851	
480	London	1857	

	Publisher	\
206	S. Tinsley & Co.	
216	Virtue & Co.	
218	Bradbury, Evans & Co.	
472	James Darling	
480	Wertheim & Macintosh	

	Title	Author	\
206	Walter Forbes. [A novel.] By A. A	A. A.	
216	All for Greed. [A novel. The dedication signed...	A., A. A.	
218	Love the Avenger. By the author of "All for Gr...	A., A. A.	
472	Welsh Sketches, chiefly ecclesiastical, to the...	A., E. S.	
480	[The World in which I live, and my place in it...	A., E. S.	

	Flickr URL
206	http://www.flickr.com/photos/britishlibrary/ta...
216	http://www.flickr.com/photos/britishlibrary/ta...
218	http://www.flickr.com/photos/britishlibrary/ta...
472	http://www.flickr.com/photos/britishlibrary/ta...
480	http://www.flickr.com/photos/britishlibrary/ta...

Technical Detail: Unlike primary keys in SQL, a Pandas Index doesn't make any guarantee of being unique, although many indexing and merging operations will notice a speedup in runtime if it is.

We can access each record in a straightforward way with `loc[]`. Although `loc[]` may not have all that intuitive of a name, it allows us to do *label-based indexing*, which is the labeling of a row or record without regard to its position:

```
Python >>>
>>> df.loc[206]
```

Place of Publication	London
Date of Publication	1879 [1878]
Publisher	S. Tinsley & Co.
Title	Walter Forbes. [A novel.] By A. A
Author	A. A.
Flickr URL	http://www.flickr.com/photos/britishlibrary/ta...
Name: 206, dtype: object	

In other words, 206 is the first label of the index. To access it by *position*, we could use `df.iloc[0]`, which does position-based indexing.

Technical Detail: `.iloc[]` is technically a [class instance](#) and has some special [syntax](#) that doesn't conform exactly to most plain-vanilla Python instance methods.

Previously, our index was a `RangeIndex`: integers starting from 0, analogous to Python's built-in `range`. By passing a column name to `set_index`, we have changed the index to the values in `Identifier`.

You may have noticed that we reassigned the variable to the object returned by the method with `df = df.set_index(...)`. This is because, by default, the method returns a modified copy of our object and does not make the changes directly to the object. We can avoid this by setting the `inplace` parameter:

Python

```
df.set_index('Identifier', inplace=True)
```

Tidying up Fields in the Data

So far, we have removed unnecessary columns and changed the index of our `DataFrame` to something more sensible. In this section, we will clean specific columns and get them to a uniform format to get a better understanding of the dataset and enforce consistency. In particular, we will be cleaning `Date of Publication` and `Place of Publication`.

Upon inspection, all of the data types are currently the object `dtype`, which is roughly analogous to `str` in native Python.

It encapsulates any field that can't be neatly fit as numerical or categorical data. This makes sense since we're working with data that is initially a bunch of messy strings:

Python

```
>>> df.get_dtype_counts()
object    6
```

One field where it makes sense to enforce a numeric value is the date of publication so that we can do calculations down the road:

Python

```
>>> df.loc[1905:, 'Date of Publication'].head(10)
Identifier
1905      1888
1929    1839, 38-54
2836      [1897?]
2854      1865
2956    1860-63
2957      1873
3017      1866
3131      1899
4598      1814
4884      1820
Name: Date of Publication, dtype: object
```

A particular book can have only one date of publication. Therefore, we need to do the following:

- Remove the extra dates in square brackets, wherever present: 1879 [1878]

- Convert date ranges to their “start date”, wherever present: 1860-63; 1839, 38-54
- Completely remove the dates we are not certain about and replace them with NumPy’s NaN: [1897?]
- Convert the string nan to NumPy’s NaN value

Synthesizing these patterns, we can actually take advantage of a single regular expression to extract the publication year:

```
Python >>>
regex = r'^(\d{4})'
```

The regular expression above is meant to find any four digits at the beginning of a string, which suffices for our case. The above is a *raw string* (meaning that a backslash is no longer an escape character), which is standard practice with regular expressions.

The `\d` represents any digit, and `{4}` repeats this rule four times. The `^` character matches the start of a string, and the parentheses denote a capturing group, which signals to Pandas that we want to extract that part of the regex. (We want `^` to avoid cases where `[` starts off the string.)

Let’s see what happens when we run this regex across our dataset:

```
Python >>>
>>> extr = df['Date of Publication'].str.extract(r'^(\d{4})', expand=False)
>>> extr.head()
Identifier
206      1879
216      1868
218      1869
472      1851
480      1857
Name: Date of Publication, dtype: object
```

Not familiar with regex? You can [inspect the expression above](https://www.regexp101.com) at [regex101.com](https://www.regexp101.com) and read more at the [Python Regular Expressions HOWTO](#).

Technically, this column still has object dtype, but we can easily get its numerical version with `pd.to_numeric`:

```
Python >>>
>>> df['Date of Publication'] = pd.to_numeric(extr)
>>> df['Date of Publication'].dtype
dtype('float64')
```

This results in about one in every ten values being missing, which is a small price to pay for now being able to do computations on the remaining valid values:

```
Python >>>
>>> df['Date of Publication'].isnull().sum() / len(df)
0.11717147339205986
```

Great! That’s done!

Combining str Methods with NumPy to Clean Columns

Above, you may have noticed the use of `df['Date of Publication'].str`. This attribute is a way to access speedy [string operations](#) in Pandas that largely mimic operations on native Python strings or compiled regular expressions, such as `.split()`, `.replace()`, and `.capitalize()`.

To clean the `Place of Publication` field, we can combine Pandas `str` methods with NumPy's `np.where` function, which is basically a vectorized form of Excel's `IF()` macro. It has the following syntax:

Python

>>>

```
>>> np.where(condition, then, else)
```

Here, `condition` is either an array-like object or a boolean mask. `then` is the value to be used if `condition` evaluates to `True`, and `else` is the value to be used otherwise.

Essentially, `.where()` takes each element in the object used for `condition`, checks whether that particular element evaluates to `True` in the context of the condition, and returns an `ndarray` containing `then` or `else`, depending on which applies.

It can be nested into a compound if-then statement, allowing us to compute values based on multiple conditions:

Python

>>>

```
>>> np.where(condition1, x1,
            np.where(condition2, x2,
                      np.where(condition3, x3, ...)))
```

We'll be making use of these two functions to clean `Place of Publication` since this column has string objects. Here are the contents of the column:

Python

>>>

```
>>> df['Place of Publication'].head(10)
Identifier
206                London
216      London; Virtue & Yorston
218                London
472                London
480                London
481                London
519                London
667      pp. 40. G. Bryan & Co: Oxford, 1898
874                London]
1143               London
Name: Place of Publication, dtype: object
```

We see that for some rows, the place of publication is surrounded by other unnecessary information. If we were to look at more values, we would see that this is the case for only some rows that have their place of publication as 'London' or 'Oxford'.

Let's take a look at two specific entries:

Python

>>>


```

>>> df.loc[4157862]
Place of Publication      Newcastle-upon-Tyne
Date of Publication      1867
Publisher                 T. Fordyce
Title                    Local Records; or, Historical Register of rema...
Author                   T. Fordyce
Flickr URL               http://www.flickr.com/photos/britishlibrary/ta...
Name: 4157862, dtype: object

>>> df.loc[4159587]
Place of Publication      Newcastle upon Tyne
Date of Publication      1834
Publisher                 Mackenzie & Dent
Title                    An historical, topographical and descriptive v...
Author                   E. (Eneas) Mackenzie
Flickr URL               http://www.flickr.com/photos/britishlibrary/ta...
Name: 4159587, dtype: object

```

These two books were published in the same place, but one has hyphens in the name of the place while the other does not.

To clean this column in one sweep, we can use `str.contains()` to get a boolean mask.

We clean the column as follows:

```

Python >>>
>>> pub = df['Place of Publication']
>>> london = pub.str.contains('London')
>>> london[:5]
Identifier
206      True
216      True
218      True
472      True
480      True
Name: Place of Publication, dtype: bool

>>> oxford = pub.str.contains('Oxford')

```

We combine them with `np.where()`:

```

Python >>>
df['Place of Publication'] = np.where(london, 'London',
                                     np.where(oxford, 'Oxford',
                                               pub.str.replace('-', ' ')))

>>> df['Place of Publication'].head()
Identifier
206      London
216      London
218      London
472      London
480      London
Name: Place of Publication, dtype: object

```

Here, the `np.where()` function is called in a nested structure, with condition being a series of booleans obtained with `str.contains()`. The `contains()` method works similarly to the built-in `in` keyword used to find the occurrence of an entity in an iterable (or substring in a string).

The replacement to be used is a string representing our desired place of publication. We also replace hyphens with a space with `str.replace()` and reassign to the column in our `DataFrame`.

Although there is more dirty data in this dataset, we will discuss only these two columns for now.

Let's have a look at the first five entries, which look a lot crisper than when we started out:

```
Python >>> df.head()
```

	Place of Publication	Date of Publication	Publisher \
206	London	1879	S. Tinsley & Co.
216	London	1868	Virtue & Co.
218	London	1869	Bradbury, Evans & Co.
472	London	1851	James Darling
480	London	1857	Wertheim & Macintosh

	Title	Author \
206	Walter Forbes. [A novel.] By A. A	AA
216	All for Greed. [A novel. The dedication signed...	A. A A.
218	Love the Avenger. By the author of "All for Gr...	A. A A.
472	Welsh Sketches, chiefly ecclesiastical, to the...	E. S A.
480	[The World in which I live, and my place in it...	E. S A.

	Flickr URL
206	http://www.flickr.com/photos/britishlibrary/ta...
216	http://www.flickr.com/photos/britishlibrary/ta...
218	http://www.flickr.com/photos/britishlibrary/ta...
472	http://www.flickr.com/photos/britishlibrary/ta...
480	http://www.flickr.com/photos/britishlibrary/ta...

Note: At this point, `Place of Publication` would be a good candidate for conversion to a `Categorical dtype`, because we can encode the fairly small unique set of cities with integers. (*The memory usage of a Categorical is proportional to the number of categories plus the length of the data; an object dtype is a constant times the length of the data.*)

Cleaning the Entire Dataset Using the `applymap` Function

In certain situations, you will see that the “dirt” is not localized to one column but is more spread out.

There are some instances where it would be helpful to apply a customized function to each cell or element of a `DataFrame`. Pandas `.applymap()` method is similar to the in-built `map()` function and simply applies a function to all the elements in a `DataFrame`.

Let's look at an example. We will create a `DataFrame` out of the “`university_towns.txt`” file:

Shell

```
$ head Datasets/univerisity_towns.txt
Alabama[edit]
Auburn (Auburn University)[1]
Florence (University of North Alabama)
Jacksonville (Jacksonville State University)[2]
Livingston (University of West Alabama)[2]
Montevallo (University of Montevallo)[2]
Troy (Troy University)[2]
Tuscaloosa (University of Alabama, Stillman College, Shelton State)[3][4]
Tuskegee (Tuskegee University)[5]
Alaska[edit]
```

We see that we have periodic state names followed by the university towns in that state: StateA TownA1 TownA2 StateB TownB1 TownB2 If we look at the way state names are written in the file, we'll see that all of them have the “[edit]” substring in them.

We can take advantage of this pattern by creating a *list of (state, city) tuples* and wrapping that list in a DataFrame:

```
Python >>>
>>> university_towns = []
>>> with open('Datasets/univerisity_towns.txt') as file:
...     for line in file:
...         if '[edit]' in line:
...             # Remember this `state` until the next is found
...             state = line
...         else:
...             # Otherwise, we have a city; keep `state` as last-seen
...             university_towns.append((state, line))

>>> university_towns[:5]
[('Alabama[edit]\n', 'Auburn (Auburn University)[1]\n'),
 ('Alabama[edit]\n', 'Florence (University of North Alabama)\n'),
 ('Alabama[edit]\n', 'Jacksonville (Jacksonville State University)[2]\n'),
 ('Alabama[edit]\n', 'Livingston (University of West Alabama)[2]\n'),
 ('Alabama[edit]\n', 'Montevallo (University of Montevallo)[2]\n')]
```

We can wrap this list in a DataFrame and set the columns as “State” and “RegionName”. Pandas will take each element in the list and set State to the left value and RegionName to the right value.

The resulting DataFrame looks like this:

```
Python >>>
>>> towns_df = pd.DataFrame(university_towns,
...                           columns=['State', 'RegionName'])

>>> towns_df.head()
   State                                     RegionName
0  Alabama[edit]\n          Auburn (Auburn University)[1]\n
1  Alabama[edit]\n          Florence (University of North Alabama)\n
2  Alabama[edit]\n  Jacksonville (Jacksonville State University)[2]\n
3  Alabama[edit]\n          Livingston (University of West Alabama)[2]\n
4  Alabama[edit]\n          Montevallo (University of Montevallo)[2]\n
```

While we could have cleaned these strings in the for loop above, Pandas makes it easy. We only need the state name and the town name and can remove everything else. While we could use Pandas’ .str() methods again here, we could also use applymap() to map a Python callable to each element of the DataFrame.

We have been using the term *element*, but what exactly do we mean by it? Consider the following “toy” DataFrame:

```
Python >>>
      0      1
0  Mock  Dataset
1 Python Pandas
2  Real  Python
3  NumPy  Clean
```

In this example, each cell (‘Mock’, ‘Dataset’, ‘Python’, ‘Pandas’, etc.) is an element. Therefore, `applymap()` will apply a function to each of these independently. Let’s define that function:

```
Python >>>
>>> def get_citystate(item):
...     if '(' in item:
...         return item[:item.find('(')]
...     elif '[' in item:
...         return item[:item.find('[')]
...     else:
...         return item
```

Pandas’ `.applymap()` only takes one parameter, which is the function (callable) that should be applied to each element:

```
Python >>>
>>> towns_df = towns_df.applymap(get_citystate)
```

First, we define a Python function that takes an element from the DataFrame as its parameter. Inside the function, checks are performed to determine whether there’s a (or [in the element or not.

Depending on the check, values are returned accordingly by the function. Finally, the `applymap()` function is called on our object. Now the DataFrame is much neater:

```
Python >>>
>>> towns_df.head()
   State  RegionName
0  Alabama      Auburn
1  Alabama      Florence
2  Alabama  Jacksonville
3  Alabama      Livingston
4  Alabama      Montevallo
```

The `applymap()` method took each element from the DataFrame, passed it to the function, and the original value was replaced by the returned value. It’s that simple!

Technical Detail: While it is a convenient and versatile method, `.applymap` can have significant runtime for larger datasets, because it maps a Python callable to each individual element. In some cases, it can be more efficient to do *vectorized* operations that utilize Cython or NumPY (which, in turn, makes calls in C) under the hood.

Renaming Columns and Skipping Rows

Often, the datasets you’ll work with will have either column names that are not easy to understand, or unimportant information in the first few and/or last rows, such as definitions of the terms in the dataset, or footnotes.

In that case, we'd want to rename columns and skip certain rows so that we can drill down to necessary information with correct and sensible labels.

To demonstrate how we can go about doing this, let's first take a glance at the initial five rows of the "olympics.csv" dataset:

```
Shell
$ head -n 5 Datasets/olympics.csv
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
,? Summer,01 !,02 !,03 !,Total,? Winter,01 !,02 !,03 !,Total,? Games,01 !,02 !,03 !,Combined total
Afghanistan (AFG),13,0,0,2,2,0,0,0,0,13,0,0,2,2
Algeria (ALG),12,5,2,8,15,3,0,0,0,15,5,2,8,15
Argentina (ARG),23,18,24,28,70,18,0,0,0,41,18,24,28,70
```

Now, we'll read it into a Pandas DataFrame:

```
Python >>>
>>> olympics_df = pd.read_csv('Datasets/olympics.csv')
>>> olympics_df.head()
   0      1      2      3      4      5      6      7      8  \
0   NaN  ? Summer 01 ! 02 ! 03 ! Total ? Winter 01 ! 02 !
1  Afghanistan (AFG) 13  0  0  2  2      0  0  0
2   Algeria (ALG) 12  5  2  8 15      3  0  0
3  Argentina (ARG) 23 18 24 28 70     18  0  0
4   Armenia (ARM)  5  1  2  9 12      6  0  0

   9      10      11      12      13      14      15
0 03 ! Total ? Games 01 ! 02 ! 03 ! Combined total
1  0  0 13  0  0  2      2
2  0  0 15  5  2  8     15
3  0  0 41 18 24 28     70
4  0  0 11  1  2  9     12
```

This is messy indeed! The columns are the string form of integers indexed at 0. The row which should have been our header (i.e. the one to be used to set the column names) is at `olympics_df.iloc[0]`. This happened because our CSV file starts with 0, 1, 2, ..., 15.

Also, if we were to go to the [source of this dataset](#), we'd see that NaN above should really be something like "Country", ? Summer is supposed to represent "Summer Games", 01 ! should be "Gold", and so on.

Therefore, we need to do two things:

- Skip one row and set the header as the first (0-indexed) row
- Rename the columns

We can skip rows and set the header while reading the CSV file by passing some parameters to the `read_csv()` function.

This function takes *a lot* of optional parameters, but in this case we only need one (`header`) to remove the 0th row:

```
Python >>>
```

```
>>> olympics_df = pd.read_csv('Datasets/olympics.csv', header=1)
>>> olympics_df.head()
```

	Unnamed: 0	? Summer	01 !	02 !	03 !	Total	? Winter	\
0	Afghanistan (AFG)		13	0	0	2	2	0
1	Algeria (ALG)		12	5	2	8	15	3
2	Argentina (ARG)		23	18	24	28	70	18
3	Armenia (ARM)		5	1	2	9	12	6
4	Australasia (ANZ) [ANZ]		2	3	4	5	12	0

	01 !.1	02 !.1	03 !.1	Total.1	? Games	01 !.2	02 !.2	03 !.2	\
0	0	0	0	0	13	0	0	2	
1	0	0	0	0	15	5	2	8	
2	0	0	0	0	41	18	24	28	
3	0	0	0	0	11	1	2	9	
4	0	0	0	0	2	3	4	5	

	Combined total
0	2
1	15
2	70
3	12
4	12

We now have the correct row set as the header and all unnecessary rows removed. Take note of how Pandas has changed the name of the column containing the name of the countries from NaN to Unnamed: 0.

To rename the columns, we will make use of a DataFrame's `rename()` method, which allows you to relabel an axis based on a *mapping* (in this case, a dict).

Let's start by defining a dictionary that maps current column names (as keys) to more usable ones (the dictionary's values):

```
Python >>> new_names = {'Unnamed: 0': 'Country',
...                  '? Summer': 'Summer Olympics',
...                  '01 !': 'Gold',
...                  '02 !': 'Silver',
...                  '03 !': 'Bronze',
...                  '? Winter': 'Winter Olympics',
...                  '01 !.1': 'Gold.1',
...                  '02 !.1': 'Silver.1',
...                  '03 !.1': 'Bronze.1',
...                  '? Games': '# Games',
...                  '01 !.2': 'Gold.2',
...                  '02 !.2': 'Silver.2',
...                  '03 !.2': 'Bronze.2'}
```

We call the `rename()` function on our object:

```
Python >>> olympics_df.rename(columns=new_names, inplace=True)
```

Setting *inplace* to True specifies that our changes be made directly to the object. Let's see if this checks out:

```
Python >>>
```

```
>>> olympics_df.head()
      Country Summer Olympics Gold Silver Bronze Total \
0  Afghanistan (AFG)          13    0     0     2     2
1    Algeria (ALG)           12    5     2     8    15
2   Argentina (ARG)          23   18    24    28    70
3    Armenia (ARM)            5    1     2     9    12
4 Australasia (ANZ) [ANZ]      2    3     4     5    12

      Winter Olympics Gold.1 Silver.1 Bronze.1 Total.1 # Games Gold.2 \
0                0      0      0      0      0      13      0
1                3      0      0      0      0      15      5
2               18      0      0      0      0      41     18
3                6      0      0      0      0      11      1
4                0      0      0      0      0      2       3

      Silver.2 Bronze.2 Combined total
0            0         2             2
1            2         8            15
2           24        28            70
3            2         9            12
4            4         5            12
```

Python Data Cleaning: Recap and Resources

In this tutorial, you learned how you can drop unnecessary information from a dataset using the `drop()` function, as well as how to set an index for your dataset so that items in it can be referenced easily.

Moreover, you learned how to clean object fields with the `.str()` accessor and how to clean the entire dataset using the `applymap()` method. Lastly, we explored how to skip rows in a CSV file and rename columns using the `rename()` method.

Knowing about data cleaning is very important, because it is a big part of data science. You now have a basic understanding of how Pandas and NumPy can be leveraged to clean datasets!

Check out the links below to find additional resources that will help you on your Python data science journey:

- The Pandas [documentation](#)
- The NumPy [documentation](#)
- [Python for Data Analysis](#) by Wes McKinney, the creator of Pandas
- [Pandas Cookbook](#) by Ted Petrou, a data science trainer and consultant

Free Bonus: [Click here to get access to a free NumPy Resources Guide](#) that points you to the best tutorials, videos, and books for improving your NumPy skills.

📧 Python Tricks 📧

Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.