

## What's Pandas for?

Pandas has so many uses that it might make sense to list the things it can't do instead of what it can do.

This tool is essentially your data's home. Through pandas, you get acquainted with your data by cleaning, transforming, and analyzing it.

For example, say you want to explore a dataset stored in a CSV on your computer. Pandas will extract the data from that CSV into a DataFrame — a table, basically — then let you do things like:

- Calculate statistics and answer questions about the data, like
  - What's the average, median, max, or min of each column?
  - Does column A correlate with column B?
  - What does the distribution of data in column C look like?
- Clean the data by doing things like removing missing values and filtering rows or columns by some criteria
- Visualize the data with help from Matplotlib. Plot bars, lines, histograms, bubbles, and more.
- Store the cleaned, transformed data back into a CSV, other file or database

Before you jump into the modeling or the complex visualizations you need to have a good understanding of the nature of your dataset and pandas is the best avenue through which to do that.

## How does pandas fit into the data science toolkit?

Not only is the pandas library a central component of the data science toolkit but it is used in conjunction with other libraries in that collection.

Pandas is built on top of the **NumPy** package, meaning a lot of the structure of NumPy is used or replicated in Pandas. Data in pandas is often used to feed statistical analysis in **SciPy**, plotting functions from **Matplotlib**, and machine learning algorithms in **Scikit-learn**.

Jupyter Notebooks offer a good environment for using pandas to do data exploration and modeling, but pandas can also be used in text editors just as easily.

Jupyter Notebooks give us the ability to execute code in a particular cell as opposed to running the entire file. This saves a lot of time when working with large datasets and complex transformations. Notebooks also provide an easy way to visualize pandas' DataFrames and plots. As a matter of fact, this article was created entirely in a Jupyter Notebook.

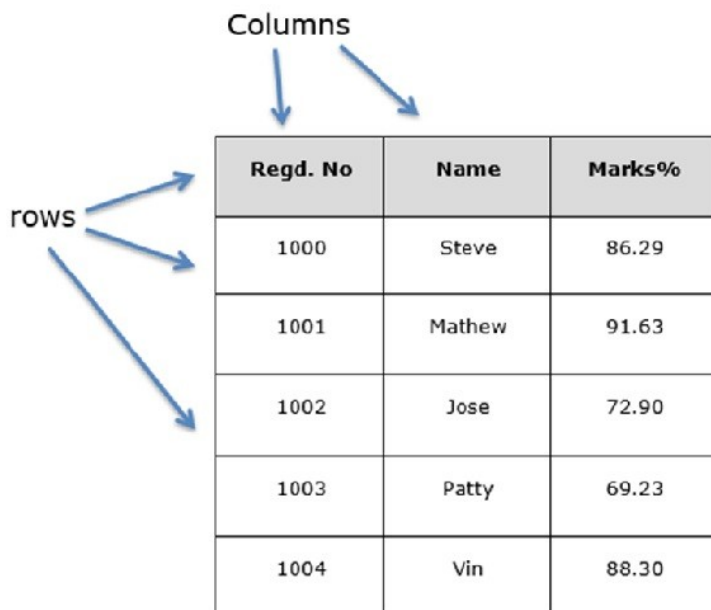
## When should you start using pandas?

If you do not have any experience coding in Python, then you should stay away from learning pandas until you do. You don't have to be at the level of the software engineer, but you should be adept at the basics, such as lists, tuples, dictionaries, functions, and iterations. Also, I'd also recommend familiarizing yourself with **NumPy** due to the similarities mentioned above.

## Pandas First Steps

### Structure

Let us assume that we are creating a data frame with student's data.



Regd. No	Name	Marks%
1000	Steve	86.29
1001	Mathew	91.63
1002	Jose	72.90
1003	Patty	69.23
1004	Vin	88.30

You can think of it as an SQL table or a spreadsheet data representation.

### Create an Empty DataFrame

A basic DataFrame, which can be created is an Empty Dataframe.

#### Example

```
#import the pandas library and aliasing as pd
import pandas as pd
df = pd.DataFrame()
print (df)
```

Its **output** is as follows –

```
Empty DataFrame
Columns: []
Index: []
```

### Create a DataFrame from Lists

The DataFrame can be created using a single list or a list of lists.

#### Example 1

```
import pandas as pd
data = [1,2,3,4,5]
df = pd.DataFrame(data)
print (df)
```

Its **output** is as follows –

	0
0	1
1	2
2	3
3	4
4	5

### Example 2

```
import pandas as pd
data = [['Alex',10],['Bob',12],['Clarke',13]]
df = pd.DataFrame(data,columns=['Name','Age'])
print(df)
```

Its **output** is as follows –

	Name	Age
0	Alex	10
1	Bob	12
2	Clarke	13

Creating a Dataframe from a CSV.

```
df = pandas.read_csv("school.csv") # change to your file path
print(df)
```

### Install and import

Pandas is an easy package to install. Open up your terminal program (for Mac users) or command line (for PC users) and install it using either of the following commands:

```
conda install pandas
```

OR

```
pip install pandas
```

Alternatively, if you're currently viewing this lesson in a Jupyter notebook you can run this cell:

```
!pip install pandas
```

The ! at the beginning runs cells as if they were in a terminal.

To import pandas we usually import it with a shorter name since it's used so much:

```
import pandas as pd
```

### Read data from csv

```
df = pd.read_csv("school.csv")
```

Now to the basic components of pandas.

Another fast and useful attribute is `.shape`, which outputs just a tuple of (rows, columns):

```
print(df.shape)
```

Out:

```
(435, 23)
```

Note that `.shape` has no parentheses and is a simple tuple of format (rows, columns). So we have 453 **rows** and 23 **columns** in our student DataFrame.

You'll be going to `.shape` a lot when cleaning and transforming data. For example, you might filter some rows based on some criteria and then want to know quickly how many rows were removed.

### How to work with missing values

When exploring data, you'll most likely encounter missing or null values, which are essentially placeholders for non-existent values. Most commonly you'll see Python's `NONE` or NumPy's `np.nan`, each of which are handled differently in some situations.

There are two options in dealing with nulls:

1. Get rid of rows or columns with nulls
2. Replace nulls with non-null values, a technique known as **imputation**

To count the number of nulls in each column we use an aggregate function for summing:

```
print(df.isnull().sum())
```

Out:

ids	0
bday	8
enrolldate	29
expgradate	29
Rank	29
Major	159
Gender	9
Athlete	0
Height	27
Weight	59
Smoking	24
Sprint	61
MileMinDur	43
English	26
Reading	10
Math	13
Writing	31
State	27
LiveOnCampus	21

```
HowCommute      188
CommuteTime     188
SleepTime       32
StudyTime       42
dtype: int64
```

`.isnull()` just by itself isn't very useful, and is usually used in conjunction with other methods, like `sum()`.

We can see now that our data has missing value in different columns.

### Removing null values

Data Scientists and Analysts regularly face the dilemma of dropping or imputing null values, and is a decision that requires intimate knowledge of your data and its context. Overall, removing null data is only suggested if you have a small amount of missing data,

Remove nulls is pretty simple:

```
df.dropna()
```

This operation will delete any **row** with at least a single null value, but it will return a new DataFrame without altering the original one. You could specify `inplace=True` in this method as well.

```
df = pd.read_csv("school.csv")
df.dropna(inplace=True)
print(df.isnull().sum())
```

Now we have; all rows with null values eliminated

```
ids            0
bday           0
enrolldate     0
expgradate     0
Rank           0
Major          0
Gender         0
Athlete        0
Height         0
Weight         0
Smoking        0
Sprint         0
MileMinDur     0
English        0
Reading        0
Math           0
Writing        0
State          0
LiveOnCampus   0
HowCommute     0
CommuteTime    0
SleepTime      0
```

```
StudyTime      0
dtype: int64
```

Lets see how many rows we lost from missing data.

```
print(df.shape)
```

Out:

```
(53, 23)
```

We lost over 75% of data!

So in the case of our dataset, this operation would remove empty rows. This obviously seems like a waste since there's perfectly good data in the other columns of those dropped rows. That's why we'll look at **imputation next**.

Other than just dropping rows, you can also drop columns with null values by setting `axis=1`:

```
df.dropna(axis=1)
```

## Imputation

Imputation is a conventional feature engineering technique used to keep valuable data that have null values.

There may be instances where dropping every row with a null value removes too big a chunk from your dataset, so instead we can impute that null with another value, usually the **mean** or the **median** of that column, or unknown for ordinal data. I.e Male, Female, *Unknown*

3. Let's look at imputing the missing values in the **English** column, which had 26 missing, instead of dropping we can fill the missing with median.
4. First we'll extract that column into its own variable and finds its median or mean. Then fill the null values for English with the median

```
medianEng = df['English'].median() # find median
```

```
df['English'].fillna(medianEng, inplace=True) # fill null with median
```

Imputing an entire column with the same value like this is a basic example. It would be a better idea to try a more granular imputation by **Math** or **Reading**.

For example, you would find the median of the `Math` and impute the nulls in each genre with that `Math` mean.

**TRY:** impute `CommuteTime`, `Math`

Let's now look at more ways to examine and understand the dataset.

That was `fillna()` for numerical columns, Now let's look at Ordinal Columns Like **Gender**. Gender had 9 missing records. Let's impute them. NB: Male was coded with 0, Female was coded with 1, so we code nulls with 2

```
df['Gender'].fillna(2, inplace=True)
```

This will fill empties in Gender with 2

**Try impute for: Rank, Smoking**

**Replacing categorical variables coded with 0, 1, 2 to Labeled names**

In our dataset Gender is coded as 0 and 1, we would want to label back to Male and Female.

```
df['Gender'].replace(0, "Male", inplace=True)
df['Gender'].replace(1, "Female", inplace=True)
df['Gender'].replace(2, "Unknown", inplace=True)
```

Try encode for `HowCommute`, `Smoking`, `Rank`.

**Understanding your variables**

Using `describe()` on an entire DataFrame we can get a summary of the distribution of continuous variables:

```
print(df.describe())
```

Understanding which numbers are continuous also comes in handy when thinking about the type of plot to use to represent your data visually.

`.describe()` can also be used on a categorical variable to get the count of rows, unique count of categories, top category, and freq of top category:

```
print(df['English'].describe())
```

Out:

```
count    409.000000
mean      82.787555
std       6.839803
min       59.830000
```

```
25%    78.330000
50%    83.150000
75%    87.170000
max    101.950000
```

Name: English, dtype: float64

This tells us that the English column has 409 unique values, the top the mean is 82.8, the std is 6.8, the lowest score in english was around 59.8 and the highest score was around 101.9

`.value_counts()` can tell us the frequency of all values in a column:

```
print(df['English'].value_counts())
```

### DataFrame slicing, selecting, extracting

Up until now we've focused on some basic summaries of our data. We've learned about simple column extraction using single brackets, and we imputed null values in a column using `fillna()`. Below are the other methods of slicing, selecting, and extracting you'll need to use constantly.

It's important to note that, although many methods are the same, DataFrames and Series have different attributes, so you'll need be sure to know which type you are working with or else you will receive attribute errors.

Let's look at working with columns first.

#### By column

You already saw how to extract a column using square brackets like this:

```
print(df['Math'])
```

Out:

It will print all Maths grade

Since it's just a list, adding another column name is easy: this will narrow down by 2 columns and have a subset dataframe

```
subset = df[['Gender', 'Reading']]
```

```
print(subset)
```

	Gender	Reading
0	0.0	81.50
1	0.0	85.25
2	0.0	86.88



3	1.0	88.68
4	1.0	77.30
5	0.0	NaN
6	0.0	87.53
7	0.0	67.31
8	1.0	76.54
9	1.0	93.88

Now we'll look at getting data by rows.

### By rows

For rows, we have two options:

- `.loc` - **l**ocates by name
- `.iloc` - **l**ocates by numerical **i**ndex

### Conditional selections

We've gone over how to select columns and rows, but what if we want to make a conditional selection?

For example, what if we want to filter our students by major DataFrame to show only students taking Anthropology.

To do that, we take a column from the DataFrame and apply a Boolean condition to it. Here's an example of a Boolean condition:

```
print(df[df['Major']=='Anthropology'])
```

END.

### Next Visualization