Pandas: Getting Started

Pandas

Pandas is a Python library used for working with datasets. It does that by helping us make sense of **DataFrames**, which are a form of two-dimensional **structured data**, like a table with columns and rows. But before we can do anything else, we need to start with data in a CSV file.

CSV Files

CSV stands for Comma Separated Values, and it's a file type that allows data to be saved in a table. Data presented in a table is called **structured data**, because it adheres to the idea that there is a meaningful relationship between the columns and rows. A CSV might also show **panel data**, which is data that shows observations of the same behavior at various different times. The datasets we're using in this part of the course are all structured tables, but you'll see other arrangements of data as you move through your projects.

If you're familiar with the way that data tables look in spreadsheet applications like Excel, you might be surprised to see that raw CSV files don't look like that. If you came across a CSV file and opened it to see what it looked like, you'd see something like this:

```
property_type,department,lat,lon,area_m2,price_usd
house,Bogotá D.C,4.69,-74.048,187.0,"$330,899.98"
house,Bogotá D.C,4.695,-74.082,82.0,"$121,555.09"
house,Quindío,4.535,-75.676,235.0,"$219,474.47"
house,Bogotá D.C,4.62,-74.129,195.0,"$97,919.38"
```

Working with DataFrames

The first thing we need to do is import pandas; we'll use `pd` as an *alias* when we include it in our code.

Pandas is just a library; to get anything done, we need a dataset, too. We'll use the `read_csv` method to create a DataFrame from a CSV file.

In [ ]:
```
import pandas as pd

df = pd.read_csv("data/colombia-real-estate-1.csv")
df.head()
```

Practice

Try it yourself! Create a DataFrame called `df2` using the `colombia-real-estate-2` CSV file.

In [ ]:
```
df2 = ...
df2.head()
```

Inspecting DataFrames

Once we've created a DataFrame, we need to **inspect** it in order to see what's there. Pandas has many ways to inspect a DataFrame, but we're only going to look at three of them: `shape`, `info`, and `head`.

If we were interested in understanding the **dimensionality** of the DataFrame, we use the `df.shape` method. The code looks like this:

In [ ]:
```
df.shape
```

The `shape` output tells us that the `colombia-real-estate-1` DataFrame -- which we called `df1` -- has 3066 rows and 6 columns.

If we were trying to get a **general idea** of what the DataFrame contained, we use the `info` method. The code looks like this:

In [ ]:
```
df.info()
```

The `info` output tells us all sorts of things about the DataFrame: the number of columns, the names of the columns, the data type for each column, how many non-null rows are contained in the DataFrame.

Practice

Try it yourself! Use `info` and `shape` to explore `df2`, which you created above.

In [ ]:
```

```

If we wanted to see all the rows in our new DataFrame, we could use the `print` method. Keep in mind that the entire dataset gets printed when you use `print`, even though it only shows you the first few lines. That's not much of a problem with this particular dataset, but once you start working with much bigger datasets, printing the whole thing will cause all sorts of problems.

So instead of doing that, we'll just take a look at the first five rows by using the `head` method. The code looks like this:

In [ ]:
```
df.head()
```

By default, `head` returns the first five rows of data, but you can specify as many rows as you like. Here's what the code looks like for just the first two rows:

In [ ]:
```
print(df.head(2))
```

Practice

Try it yourself! Use the `head` method to return the first five and first 7 rows of the `colombia-real-estate-2` dataset.

In [ ]:
```
```

Working with Columns

Sometimes, it's handy to duplicate a column of data. It might be that you'd like to drop some data points or erase empty cells while still preserving the original column. If you'd like to do that, you'll need to duplicate the column. We can do this by placing the name of the new column in square brackets.

Adding Columns

For example, we might want to add a column of data that shows the price per square meter of each house in US dollars. To do that, we're going to need to create a new column, and include the necessary math to populate it. First, we need to import the CSV and inspect the first five rows using the `head` method, like this:

In [ ]:
```
df3 = pd.read_csv("data/colombia-real-estate-3.csv")
df3.head()
```

Then, we create a new column called `"price_m2"`, provide the formula to populate it, and inspect the first five rows of the dataset to make sure the new column includes the new values:

In [ ]:
```
df3["price_m2"] = df3["price_usd"] / df["area_m2"]
df3.head()
```

Practice

Try it yourself! Add a column to the `colombia-real-estate-2` dataset that shows the price per square meter of each house in Colombian pesos.

In [ ]:
```
df = ...
df["price_m2"] = ...
```

Dropping Columns

Just like we can add columns, we can also take them away. To do this, we'll use the `drop` method. If I wanted to drop the `"department"` column from `colombia-real-estate-1`, the code would look like this:

In [ ]:
```
df2 = df.drop("department", axis="columns")
df2.head()
```

Note that we specified that we wanted to drop a column by setting the `axis` argument to `"columns"`. We can drop rows from the dataset if we change the `axis` argument to `"index"`. If we wanted to drop row 2 from the `df2` data, the code would look like this:

In [ ]:
```
df2 = df.drop(2, axis="index")
df2.head()
```

Practice

Try it yourself! Drop the `"property_type"` column and row 4 in the `colombia-real-estate-2` dataset.

In [ ]:
```
df1 = ...
```

Dropping Rows

Including rows with empty cells can radically skew the results of our analysis, so we often drop them from the dataset. We can do this with the `dropna` method. If we wanted to do this with `df`, the code would look like this:

In [ ]:
```
print("df shape before dropping rows", df.shape)
df.dropna(inplace=True)
print("df shape after dropping rows", df.shape)
df.head()
```

By default, pandas will keep the original DataFrame, and will create a copy that reflects the changes we just made. That's perfectly fine, but if we want to make sure that copies of the DataFrame aren't clogging up the memory on our computers, then we need to intervene with the `inplace` argument. `inplace=True` means that we want the original DataFrame updated without making a copy. If we don't include `inplace=True` (or if we do include `inplace=False`), then pandas will revert to the default.

Practice

Drop rows with empty cells from the `colombia-real-estate-2` dataset.

In [ ]:
```
df2 = ...
```

Splitting Strings

It might be useful to split strings into their constituent parts, and create new columns to contain them. To do this, we'll use the `.str.split` method, and include the character we want to use as the place where the data splits apart. In the `colombia-real-estate-3` dataset, we might be

interested breaking the `"lat-lon"` column into a `"lat"` column and a `"lon"` column. We'll split it at `","` with code that looks like this:

In [ ]:
```
df3[["lat", "lon"]] = df3["lat-lon"].str.split(",", expand=True)
```

Here, `expand` is telling pandas to make the DataFrame bigger; that is, to create a new column without dropping any of the ones that already exist.

Practice

Try it yourself! In `df3`, split `"place_with_parent_names"` into three columns (one called `"place"`, one called `"department"`, and one called `"state"`, using the character `"|"`, and then return the new `"department"` column.

In [ ]:
```

```

Recasting Data

Depending on who formatted your dataset, the types of data assigned to each column might need to be changed. If, for example, a column containing only numbers had been mistaken for a column containing only strings, we'd need to change that through a process called *recasting*. Using the `colombia-real-estate-1` dataset, we could recast the entire dataset as strings by using the `astype` method, like this:

In [ ]:
```
print(df.info())
newdf = df.astype("str")
print(newdf.info())
```

This is a useful approach, but, more often than not, you'll want to only recast individual columns. In the `colombia-real-estate-1` dataset, the `"area_m2"` column is cast as `float64`. Let's change it to `int`. We'll still use the `astype` method, but we'll insert the name of the column. The code looks like this:

In [ ]:
```
df["area_m2"] = df.area_m2.astype(int)
df.info()
```

Practice

Try it yourself! In the `colombia-real-estate-2` dataset, recast `"price_cop"` as an object.

In [ ]:
```
df = ...
df2["price_cop"] = ...
df.info()
```

Access a substring in a Series

To access a substring from a Series, use the `.str` attribute from the Series. Then, index each string in the Series by providing the `start:stop:step`. Keep in mind that the start position is inclusive and the stop position is exclusive, meaning the value at the start index is included but the value at the stop index is not included. Also, Python is a 0-indexed language, so the first element in the substring is at index position 0. For example, using the `colombia-real-estate-1` dataset, we could the values at index position 0, 2, and 4 of the `department` column:

In [ ]:
```
df["department"].str[0:5:2]
```

Practice: Access a substring in a Series using pandas

Try it yourself! In the `colombia-real-estate-2` dataset, access the `property_type` column and return the first 5 characters from each row:

In [ ]:
```

```

Replacing String Characters

Another change you might want to make is replacing the characters in a string. To do this, we'll use the `replace` method again, being sure to specify which string should be replaced, and what new string should replace it. For example, if we wanted to replace the string `"house"` with the string `"single_family"` in the `colombia-real-estate-1` dataset, the code would look like this:

In [ ]:
```
df["property_type"] = df["property_type"].str.replace("house", "single_family")
df.head()
```

There are two important things to note here. The first is that the old value needs to come before the new value inside the parentheses of `str.replace`.

The second important issue here is that, unless you specify differently, *all* instances of the old value will be replaced. If you only want to replace the first three instances, the code would look like this: `str.replace("house", "single_family", 3)`

In [ ]:
```
df["property_type"] = df["property_type"].str.replace("house", "single_family", 3)
df.head()
```

Practice

Try it yourself! In the `colombia-real-estate-2` dataset, change `"apartment"` to `"multi_family"`, in the first 7 rows, and print the result.

In [ ]:
```
df = ...
```

Rename a Series

Another change you might want to make is to rename a Series in pandas. To do this, we'll use the `rename` method, being sure to specify the mapping of old and new columns. For example, if we wanted to replace the column name `property_type` with the string `type_property` in the `colombia-real-estate-1` dataset, the code would look like this:

In [ ]:
```
df.rename(columns={"property_type": "type_property"})
```

Practice: Rename a Series

Try it yourself! In the `colombia-real-estate-2` dataset, change the column `lat` to `latitude` and print the head of DataFrame.

In [ ]:
```

```

Determine the unique values in a column

You might be interested in the unique values in a Series using pandas. To do this, we'll use the `unique` method. For example, if we wanted to identify the unique values in the column `property_type` in the `colombia-real-estate-1` dataset, the code would look like this:

In [ ]:
```
df["property_type"].unique()
```

Practice: Determine the unique values in a column

Try it yourself! In the `colombia-real-estate-2` dataset, identify the unique values in the column `department`:

In [ ]:
```

```

Concatenating

When we **concatenate** data, we're combining two or more separate sets of data into a single large dataset.

Concatenating DataFrames

If we want to combine two DataFrames, we need to import Pandas and read in our data.

In [ ]:
```
df1 = pd.read_csv("data/colombia-real-estate-1.csv")
df2 = pd.read_csv("data/colombia-real-estate-2.csv")
print("df1 shape:", df1.shape)
print("df2 shape:", df2.shape)
```

Next, we'll use the `concat` method to put our DataFrames together, using each DataFrame's name

in a list.

In [ ]:
```
concat_df = pd.concat([df1, df2])
print("concat_df shape:", concat_df.shape)
concat_df.head()
```

Practice

Try it yourself! Create two DataFrames from `colombia-real-estate-2.csv` and `colombia-real-estate-3.csv`, and concatenate them as the DataFrame `concat_df`.

In [ ]:
```
df2 = ...
df3 = ...
concat_df = ...
concat_df.head()
```

Concatenating Series

We can also concatenate a Series using a similar set of commands. First, let's take two Series from the `df1` and `df2` respectively.

In [ ]:
```
df1 = pd.read_csv("data/colombia-real-estate-1.csv")
df2 = pd.read_csv("data/colombia-real-estate-2.csv")
sr1 = df1["property_type"]
sr2 = df2["property_type"]
print("len sr1:", len(sr1)),
print(sr1.head())
print()
print("len sr2:", len(sr2)),
print(sr2.head())
```

Now that we have two Series, let's put them together.

In [ ]:
```
concat_sr = pd.concat([sr1, sr2])
print("len concat_sr:", len(concat_sr)),
print(concat_sr.head())
```

Practice

Try it yourself! Use the `colombia-real-estate-2` and `colombia-rea-estate-3` datasets to create a concatenated Series for the `area_m2` column, and print the result.

In [ ]:
```
df1 = ...
```

Saving a DataFrame as a CSV

Once you've cleaned all your data and gotten the DataFrame to show everything you want it to show, it's time to save the DataFrame as a new CSV file using the `to_csv` method. First, let's load up the `colombia-real-estate-1` dataset, and use `head` to see the first five rows of data:

In [ ]:
```
import pandas as pd

df = pd.read_csv("data/colombia-real-estate-1.csv")
df.head()
```

Maybe we're only interested in those first five rows, so let's save that as its own new CSV file using the `to_csv` method. Note that we're setting the `index` argument to `False` so that the DataFrame index isn't included in the CSV file.

In [ ]:
```
df = df.head()
df.to_csv("data/small-df.csv", index=False)
```

References & Further Reading

- Tutorial for `shape`
- Tutorial for `info`
- Adding columns to a DataFrame
- Dropping columns from a DataFrame
- Splitting columns in a DataFrame
- Recasting values
- Replacing strings
- Concatenating DataFrames
- From DataFrames to Series
