03-pandas-getting-started

May 20, 2022

Pandas: Getting Started

1 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.

2 Importing Data

2.1 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"
```

2.2 Dictionaries

You can create a DataFrame from a Python dictionary using `from_dict` function.

[1]:
```
import pandas as pd

data = {"col_1": [3, 2, 1, 0], "col_2": ["a", "b", "c", "d"]}
pd.DataFrame.from_dict(data)
```

[1]:
```
   col_1 col_2
0      3     a
1      2     b
2      1     c
3      0     d
```

By default, DataFrame will be created using keys as columns. Note the length of the values should be equal for each key for the code to work. We can also let keys to be index instead of the columns:

[2]:
```
pd.DataFrame.from_dict(data, orient="index")
```

[2]:
```
        0  1  2  3
col_1   3  2  1  0
col_2   a  b  c  d
```

We can also specify column names:

[3]:
```
pd.DataFrame.from_dict(data, orient="index", columns=["A", "B", "C", "D"])
```

[3]:
```
        A  B  C  D
col_1   3  2  1  0
col_2   a  b  c  d
```

Practice

Try it yourself! Create a DataFrame called using the dictionary `clothes` and make the keys as index, and put column names as [‘color’,‘size’]

[4]:
```
clothes = {"shirt": ["red", "M"], "sweater": ["yellow", "L"], "jacket":␣
↪["black", "L"]}
```

2.3 JSON Files

JSON is short for JavaScript Object Notation. It is another widely used data format to store and transfer the data. It is light-weight and very human readable. In Python, we can use the `json` library to read JSON files. Here is an example of a JSON string.

[5]:
```
info = """{
    "firstName": "Jane",
    "lastName": "Doe",
    "hobby": "running",
    "age": 35
}"""
print(info)
```

```
{
    "firstName": "Jane",
    "lastName": "Doe",
    "hobby": "running",
```

```
    "age": 35
}
```

Use `json` library to load the json string into a Python dictionary:

[6]:
```
import json as js

data = js.loads(info)
data
```

[6]:
```
{'firstName': 'Jane', 'lastName': 'Doe', 'hobby': 'running', 'age': 35}
```

We can load a json string or file into a dictionary because they are organized in the same way: key-value pairs.

[7]:
```
data["firstName"]
```

[7]:
```
'Jane'
```

A dictionary may not be as convenient as a `DataFrame` in terms of data manipulation and cleaning. But once we've turned our json string into a dictionary, we can transform it into a `DataFrame` using the `from_dict` method.

[8]:
```
df = pd.DataFrame.from_dict(data, orient="index", columns=["subject 1"])
df
```

[8]:
```
           subject 1
firstName       Jane
lastName         Doe
hobby        running
age               35
```

Practice

Try it yourself! Load the JSON file `clothes` and then transform it to `DataFrame`, name column properly.

[9]:
```
clothes = """{"shirt": ["red","M"], "sweater": ["yellow","L"]}"""

import json as js
import pandas as pd
data = js.loads(clothes)
df = pd.DataFrame(data)
df
```

[9]:
```
  shirt sweater
0   red  yellow
1     M       L
```

3 Load Compressed file in Python

In the big data era, it is very likely that we'll need to read data from compressed files. In this case, we need to use gzip to unzip the data before loading it. We can load the `poland-bankruptcy-data-2008.json.gz` file from the data folder using the following code:

[10]:
```
import gzip
import json

with gzip.open("data/poland-bankruptcy-data-2008.json.gz", "r") as f:
    poland_data_gz = json.load(f)
```

`poland_data_gz` is a dictionary, and we only need the `data` portion of it.

[11]:
```
poland_data_gz.keys()
```

[11]:
```
dict_keys(['schema', 'data', 'metadata'])
```

We can use the `from_dict` function from pandas to read the data:

[12]:
```
df = pd.DataFrame().from_dict(poland_data_gz["data"])
```

[13]:
```
df.head()
```

[13]:

	company_id	feat_1	feat_2	feat_3	feat_4	feat_5	feat_6	feat_7	\
0	1	0.202350	0.46500	0.240380	1.5171	-14.547	0.510690	0.25366	
1	2	0.030073	0.59563	0.186680	1.3382	-37.859	-0.000319	0.04167	
2	3	0.257860	0.29949	0.665190	3.2211	71.799	0.000000	0.31877	
3	4	0.227160	0.67850	0.042784	1.0828	-88.212	0.000000	0.28505	
4	5	0.085443	0.38039	0.359230	1.9444	21.731	0.187900	0.10823	

	feat_8	feat_9	…	feat_56	feat_57	feat_58	feat_59	feat_60	\
0	0.91816	1.15190	…	0.13184	0.473950	0.86816	0.00024	8.5487	
1	0.67890	0.32356	…	0.12146	0.074369	0.87235	0.00000	1.5264	
2	2.33200	1.67620	…	0.16499	0.369210	0.81614	0.00000	4.3325	
3	0.47384	1.32410	…	0.29358	0.706570	0.78617	0.48456	5.2309	
4	1.37140	1.11260	…	0.10124	0.163790	0.89876	0.00000	5.7035	

	feat_61	feat_62	feat_63	feat_64	bankrupt
0	5.16550	107.740	3.38790	5.3440	False
1	0.63305	622.660	0.58619	1.2381	False
2	3.19850	65.215	5.59690	47.4660	False
3	5.06750	142.460	2.56210	3.0066	False
4	4.00200	89.058	4.09840	5.9874	False

[5 rows x 66 columns]

Practice

Read `poland-bankruptcy-data-2007.json.gz` into a DataFrame.

[14]:
```
# Load file into dictionary
import gzip
import pandas as pd
with gzip.open("data/poland-bankruptcy-data-2007.json.gz","r") as a:



# Transform dictionary into DataFrame
     df=pd.DataFrame().from_dict(poland_data_gz["data"])
df.head()
```

[14]:

	company_id	feat_1	feat_2	feat_3	feat_4	feat_5	feat_6	feat_7	\
0	1	0.202350	0.46500	0.240380	1.5171	-14.547	0.510690	0.25366	
1	2	0.030073	0.59563	0.186680	1.3382	-37.859	-0.000319	0.04167	
2	3	0.257860	0.29949	0.665190	3.2211	71.799	0.000000	0.31877	
3	4	0.227160	0.67850	0.042784	1.0828	-88.212	0.000000	0.28505	
4	5	0.085443	0.38039	0.359230	1.9444	21.731	0.187900	0.10823	

	feat_8	feat_9	…	feat_56	feat_57	feat_58	feat_59	feat_60	\
0	0.91816	1.15190	…	0.13184	0.473950	0.86816	0.00024	8.5487	
1	0.67890	0.32356	…	0.12146	0.074369	0.87235	0.00000	1.5264	
2	2.33200	1.67620	…	0.16499	0.369210	0.81614	0.00000	4.3325	
3	0.47384	1.32410	…	0.29358	0.706570	0.78617	0.48456	5.2309	
4	1.37140	1.11260	…	0.10124	0.163790	0.89876	0.00000	5.7035	

	feat_61	feat_62	feat_63	feat_64	bankrupt
0	5.16550	107.740	3.38790	5.3440	False
1	0.63305	622.660	0.58619	1.2381	False
2	3.19850	65.215	5.59690	47.4660	False
3	5.06750	142.460	2.56210	3.0066	False
4	4.00200	89.058	4.09840	5.9874	False

[5 rows x 66 columns]

3.1 Pickle Files

Pickle in Python is primarily used in `serializing` and `deserializing` a Python object structure. `Serialization` is the process of turning an object in memory into a stream of bytes so you can store it on disk or send it over a network. `Deserialization` is the reverse process: turning a stream of bytes back into an object in memory.

According to the pickle module documentation, the following types can be pickled:

- `None`
- Booleans
- Integers, long integers, floating point numbers, complex numbers
- Normal and Unicode strings

- Tuples, lists, sets, and dictionaries containing only objects that can be pickled
- Functions defined at the top level of a module
- Built-in functions defined at the top level of a module
- Classes that are defined at the top level of a module

Let's demonstrate using a python dictionary as an example.

[15]:
```
clothes = {"shirt": ["red", "M"], "sweater": ["yellow", "L"], "jacket":␣
↪["black", "L"]}
clothes
```

[15]:
```
{'shirt': ['red', 'M'], 'sweater': ['yellow', 'L'], 'jacket': ['black', 'L']}
```

[16]:
```
import pickle

pickle.dump(clothes, open("./data/clothes.pkl", "wb"))
```

Now in the data folder, there will be a file named `clothes.pkl`. We can read the pickled file using the following code:

[17]:
```
with open("./data/clothes.pkl", "rb") as f:
    unpickled = pickle.load(f)
```

[18]:
```
unpickled
```

[18]:
```
{'shirt': ['red', 'M'], 'sweater': ['yellow', 'L'], 'jacket': ['black', 'L']}
```

Note first we are using `wb` inside the `open` function because we are creating this file, while `deserializing` the file, we are using `rb` to read the file

Practice

Store the sample list into a pickle file, and load the pickle file back to a list.

[19]:
```
sample_list = [1, 2, 3, 4, 5]
```

[20]:
```
pickle.dump(sample_list,open("./sample_list.pkl", "wb"))
with open("./sample_list.pkl","rb") as up:
    unpickled=pickle.load(up)

unpickled
```

[20]:
```
[1, 2, 3, 4, 5]
```

4 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.

[59]:
```
import pandas as pd

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

[59]:

	property_type	department	lat	lon	area_m2	price_usd
0	house	Bogotá D.C	4.690	-74.048	187.0	$330,899.98
1	house	Bogotá D.C	4.695	-74.082	82.0	$121,555.09
2	house	Quindío	4.535	-75.676	235.0	$219,474.47
3	house	Bogotá D.C	4.620	-74.129	195.0	$97,919.38
4	house	Atlántico	11.012	-74.834	112.0	$115,477.34

Practice

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

[22]:
```
df2 = pd.read_csv("data/colombia-real-estate-2.csv")
df2.head()
```

[22]:

	property_type	department	lat	lon	area_m2	price_cop
0	house	Magdalena	11.233	-74.204	235.0	4.000000e+08
1	house	Bogotá D.C	4.714	-74.030	130.0	8.500000e+08
2	house	Cundinamarca	4.851	-74.059	137.0	4.750000e+08
3	house	Atlántico	11.006	-74.808	346.0	1.400000e+09
4	house	Cundinamarca	4.857	-74.061	175.0	4.300000e+08

5 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:

[23]:
```
df.shape
```

[23]:
```
(3066, 6)
```

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:

[24]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3066 entries, 0 to 3065
Data columns (total 6 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   property_type  3066 non-null   object
 1   department     3066 non-null   object
 2   lat            2967 non-null   float64
 3   lon            2967 non-null   float64
 4   area_m2        3066 non-null   float64
 5   price_usd      3066 non-null   object
dtypes: float64(3), object(3)
memory usage: 143.8+ KB
```

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.

[25]:
```
df2.shape
df2.info
```

[25]:
```
<bound method DataFrame.info of      property_type    department        lat
lon  area_m2     price_cop
0            house     Magdalena  11.233000 -74.204000    235.0  4.000000e+08
1            house    Bogotá D.C   4.714000 -74.030000    130.0  8.500000e+08
2            house  Cundinamarca   4.851000 -74.059000    137.0  4.750000e+08
3            house     Atlántico  11.006000 -74.808000    346.0  1.400000e+09
4            house  Cundinamarca   4.857000 -74.061000    175.0  4.300000e+08
…            …           …        …         …      …           …
3061         house  Cundinamarca   5.027000 -73.969000    164.0  4.000000e+08
3062         house     Atlántico  11.007000 -74.805000    114.0  3.100000e+08
3063         house    Bogotá D.C   4.685000 -74.148000    121.0  4.800000e+08
3064         house    Bogotá D.C   4.491000 -74.116000    150.0  9.900000e+08
3065     apartment  Cundinamarca   4.683932 -74.056925     82.0  5.900000e+08

[3066 rows x 6 columns]>
```

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:

[26]:
```
df.head()
```

[26]:

	property_type	department	lat	lon	area_m2	price_usd
0	house	Bogotá D.C	4.690	-74.048	187.0	$330,899.98
1	house	Bogotá D.C	4.695	-74.082	82.0	$121,555.09
2	house	Quindío	4.535	-75.676	235.0	$219,474.47
3	house	Bogotá D.C	4.620	-74.129	195.0	$97,919.38
4	house	Atlántico	11.012	-74.834	112.0	$115,477.34

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:

[27]:
```
print(df.head(2))
```

```
  property_type  department    lat     lon  area_m2    price_usd
0         house  Bogotá D.C  4.690 -74.048    187.0  $330,899.98
1         house  Bogotá D.C  4.695 -74.082     82.0  $121,555.09
```

Practice

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

[28]:
```
df2.head(7)
```

[28]:

	property_type	department	lat	lon	area_m2	price_cop
0	house	Magdalena	11.233	-74.204	235.0	4.000000e+08
1	house	Bogotá D.C	4.714	-74.030	130.0	8.500000e+08
2	house	Cundinamarca	4.851	-74.059	137.0	4.750000e+08
3	house	Atlántico	11.006	-74.808	346.0	1.400000e+09
4	house	Cundinamarca	4.857	-74.061	175.0	4.300000e+08
5	house	Cundinamarca	4.883	-74.035	297.0	1.500000e+09
6	house	Atlántico	11.020	-74.864	170.0	5.500000e+08

6 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.

6.1 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:

[29]:
```
df3 = pd.read_csv("data/colombia-real-estate-3.csv")
df3.head()
```

[29]:
```
  property_type            place_with_parent_names         lat-lon  area_m2  \
0         house         |Colombia|Bogotá D.C|Suba|   4.722,-74.059    113.0
1         house    |Colombia|Valle del Cauca|Cali|   3.455,-76.522    210.0
2         house    |Colombia|Bogotá D.C|Chapinero|   4.676,-74.044    183.0
3         house  |Colombia|Atlántico|Barranquilla|  10.999,-74.816     85.0
4         house    |Colombia|Valle del Cauca|Cali|   3.334,-76.547    145.0

   price_usd
0  162073.45
1  151943.86
2  422066.30
3   84413.26
4  131577.65
```

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:

[30]:
```
df3["price_m2"] = df3["price_usd"] / df["area_m2"]
df3.head()
```

[30]:
```
  property_type            place_with_parent_names         lat-lon  area_m2  \
0         house         |Colombia|Bogotá D.C|Suba|   4.722,-74.059    113.0
1         house    |Colombia|Valle del Cauca|Cali|   3.455,-76.522    210.0
2         house    |Colombia|Bogotá D.C|Chapinero|   4.676,-74.044    183.0
3         house  |Colombia|Atlántico|Barranquilla|  10.999,-74.816     85.0
4         house    |Colombia|Valle del Cauca|Cali|   3.334,-76.547    145.0

   price_usd     price_m2
0  162073.45   866.702941
1  151943.86  1852.973902
2  422066.30  1796.026809
3   84413.26   432.888513
4  131577.65  1174.800446
```

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.

[31]:
```
df = pd.read_csv("data/colombia-real-estate-2.csv")
df["price_m2"] = df['price_cop']/df["area_m2"]
df.head()
```

[31]:
```
  property_type    department     lat     lon  area_m2     price_cop  \
0         house     Magdalena  11.233 -74.204    235.0  4.000000e+08
1         house    Bogotá D.C   4.714 -74.030    130.0  8.500000e+08
2         house  Cundinamarca   4.851 -74.059    137.0  4.750000e+08
3         house     Atlántico  11.006 -74.808    346.0  1.400000e+09
```

```
4         house  Cundinamarca   4.857 -74.061    175.0  4.300000e+08

       price_m2
0  1.702128e+06
1  6.538462e+06
2  3.467153e+06
3  4.046243e+06
4  2.457143e+06
```

6.2 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:

[32]:
```
df2 = df.drop("department", axis="columns")
df2.head()
```

[32]:

	property_type	lat	lon	area_m2	price_cop	price_m2
0	house	11.233	-74.204	235.0	4.000000e+08	1.702128e+06
1	house	4.714	-74.030	130.0	8.500000e+08	6.538462e+06
2	house	4.851	-74.059	137.0	4.750000e+08	3.467153e+06
3	house	11.006	-74.808	346.0	1.400000e+09	4.046243e+06
4	house	4.857	-74.061	175.0	4.300000e+08	2.457143e+06

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:

[33]:
```
df2 = df.drop(2, axis="index")
df2.head()
```

[33]:

	property_type	department	lat	lon	area_m2	price_cop	price_m2
0	house	Magdalena	11.233	-74.204	235.0	4.000000e+08	1.702128e+06
1	house	Bogotá D.C	4.714	-74.030	130.0	8.500000e+08	6.538462e+06
3	house	Atlántico	11.006	-74.808	346.0	1.400000e+09	4.046243e+06
4	house	Cundinamarca	4.857	-74.061	175.0	4.300000e+08	2.457143e+06
5	house	Cundinamarca	4.883	-74.035	297.0	1.500000e+09	5.050505e+06

Practice

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

[34]:
```
df1 = df.drop("property_type",axis="columns")
df1.head()
```

[34]:

	department	lat	lon	area_m2	price_cop	price_m2
0	Magdalena	11.233	-74.204	235.0	4.000000e+08	1.702128e+06
1	Bogotá D.C	4.714	-74.030	130.0	8.500000e+08	6.538462e+06
2	Cundinamarca	4.851	-74.059	137.0	4.750000e+08	3.467153e+06
3	Atlántico	11.006	-74.808	346.0	1.400000e+09	4.046243e+06
4	Cundinamarca	4.857	-74.061	175.0	4.300000e+08	2.457143e+06

6.3 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:

[35]:
```
print("df shape before dropping rows", df.shape)
df.dropna(inplace=True)
print("df shape after dropping rows", df.shape)
df.head()
```

```
df shape before dropping rows (3066, 7)
df shape after dropping rows (2958, 7)
```

[35]:

	property_type	department	lat	lon	area_m2	price_cop	\
0	house	Magdalena	11.233	-74.204	235.0	4.000000e+08	
1	house	Bogotá D.C	4.714	-74.030	130.0	8.500000e+08	
2	house	Cundinamarca	4.851	-74.059	137.0	4.750000e+08	
3	house	Atlántico	11.006	-74.808	346.0	1.400000e+09	
4	house	Cundinamarca	4.857	-74.061	175.0	4.300000e+08	

	price_m2
0	1.702128e+06
1	6.538462e+06
2	3.467153e+06
3	4.046243e+06
4	2.457143e+06

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.

[36]:
```
df2.dropna(inplace=True)
df2
```

[36]:

	property_type	department	lat	lon	area_m2	price_cop	price_m2
0	house	Magdalena	11.233000	-74.204000	235.0	4.000000e+08	1.702128e+06
1	house	Bogotá D.C	4.714000	-74.030000	130.0	8.500000e+08	6.538462e+06
3	house	Atlántico	11.006000	-74.808000	346.0	1.400000e+09	4.046243e+06
4	house	Cundinamarca	4.857000	-74.061000	175.0	4.300000e+08	2.457143e+06
5	house	Cundinamarca	4.883000	-74.035000	297.0	1.500000e+09	5.050505e+06
…	…	…	…	…	…	…	…
3061	house	Cundinamarca	5.027000	-73.969000	164.0	4.000000e+08	2.439024e+06
3062	house	Atlántico	11.007000	-74.805000	114.0	3.100000e+08	2.719298e+06
3063	house	Bogotá D.C	4.685000	-74.148000	121.0	4.800000e+08	3.966942e+06
3064	house	Bogotá D.C	4.491000	-74.116000	150.0	9.900000e+08	6.600000e+06
3065	apartment	Cundinamarca	4.683932	-74.056925	82.0	5.900000e+08	7.195122e+06

[2957 rows x 7 columns]

[37]:
```
df3
```

[37]:

	property_type	place_with_parent_names	lat-lon
0	house	|Colombia|Bogotá D.C|Suba|	4.722,-74.059
1	house	|Colombia|Valle del Cauca|Cali|	3.455,-76.522
2	house	|Colombia|Bogotá D.C|Chapinero|	4.676,-74.044
3	house	|Colombia|Atlántico|Barranquilla|	10.999,-74.816
4	house	|Colombia|Valle del Cauca|Cali|	3.334,-76.547
…	…	…	…
3060	house	|Colombia|Bogotá D.C|Usaquén|	4.728,-74.044
3061	house	|Colombia|Bogotá D.C|Usaquén|	4.682,-74.056
3062	house	|Colombia|Valle del Cauca|Cali|	3.346,-76.537
3063	house	|Colombia|Bogotá D.C|Usaquén|	4.7,-74.028
3064	house	|Colombia|Bogotá D.C|Suba|	4.718,-74.08

```
      area_m2  price_usd     price_m2
0       113.0  162073.45   866.702941
1       210.0  151943.86  1852.973902
2       183.0  422066.30  1796.026809
3        85.0   84413.26   432.888513
4       145.0  131577.65  1174.800446
…         …         …           …
3060    200.0  189085.70  1139.070482
3061     66.0  191854.45   845.173789
3062    330.0  330899.98  1272.692231
3063    223.0  489596.90  5627.550575
3064    262.0  506479.56  6753.060800

[3065 rows x 6 columns]
```

6.4 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:

[38]:
```
df3[["lat", "lon"]] = df3["lat-lon"].str.split(",", expand=True)
df3.head()
```

[38]:
```
  property_type            place_with_parent_names         lat-lon  area_m2  \
0         house         |Colombia|Bogotá D.C|Suba|   4.722,-74.059    113.0
1         house    |Colombia|Valle del Cauca|Cali|   3.455,-76.522    210.0
2         house    |Colombia|Bogotá D.C|Chapinero|   4.676,-74.044    183.0
3         house  |Colombia|Atlántico|Barranquilla|  10.999,-74.816     85.0
4         house    |Colombia|Valle del Cauca|Cali|   3.334,-76.547    145.0

   price_usd     price_m2     lat      lon
0  162073.45   866.702941   4.722  -74.059
1  151943.86  1852.973902   3.455  -76.522
2  422066.30  1796.026809   4.676  -74.044
3   84413.26   432.888513  10.999  -74.816
4  131577.65  1174.800446   3.334  -76.547
```

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.

[39]:
```
df3[["","place","department","state"]]=df3["place_with_parent_names"].str.
 ↪split("|",n=3,expand=True)
df3.head()
df3['department']
```

[39]:
```
0            Bogotá D.C
1       Valle del Cauca
2            Bogotá D.C
3             Atlántico
4       Valle del Cauca
             …
3060         Bogotá D.C
3061         Bogotá D.C
3062    Valle del Cauca
3063         Bogotá D.C
3064         Bogotá D.C
Name: department, Length: 3065, dtype: object
```

6.5 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:

[40]:
```
print(df.info())
newdf = df.astype("str")
print(newdf.info())
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 2958 entries, 0 to 3065
Data columns (total 7 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   property_type  2958 non-null   object
 1   department     2958 non-null   object
 2   lat            2958 non-null   float64
 3   lon            2958 non-null   float64
 4   area_m2        2958 non-null   float64
 5   price_cop      2958 non-null   float64
 6   price_m2       2958 non-null   float64
dtypes: float64(5), object(2)
memory usage: 184.9+ KB
None
<class 'pandas.core.frame.DataFrame'>
Int64Index: 2958 entries, 0 to 3065
Data columns (total 7 columns):
```

```
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   property_type  2958 non-null   object
 1   department     2958 non-null   object
 2   lat            2958 non-null   object
 3   lon            2958 non-null   object
 4   area_m2        2958 non-null   object
 5   price_cop      2958 non-null   object
 6   price_m2       2958 non-null   object
dtypes: object(7)
memory usage: 184.9+ KB
None
```

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:

[41]:
```
df["area_m2"] = df.area_m2.astype(int)
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 2958 entries, 0 to 3065
Data columns (total 7 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   property_type  2958 non-null   object
 1   department     2958 non-null   object
 2   lat            2958 non-null   float64
 3   lon            2958 non-null   float64
 4   area_m2        2958 non-null   int64
 5   price_cop      2958 non-null   float64
 6   price_m2       2958 non-null   float64
dtypes: float64(4), int64(1), object(2)
memory usage: 184.9+ KB
```

Practice

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

[42]:
```
df2
```

[42]:

	property_type	department	lat	lon	area_m2	price_cop	\
0	house	Magdalena	11.233000	-74.204000	235.0	4.000000e+08	
1	house	Bogotá D.C	4.714000	-74.030000	130.0	8.500000e+08	
3	house	Atlántico	11.006000	-74.808000	346.0	1.400000e+09	
4	house	Cundinamarca	4.857000	-74.061000	175.0	4.300000e+08	
5	house	Cundinamarca	4.883000	-74.035000	297.0	1.500000e+09	
…	…	…	…	…	…	…	

```
3061         house  Cundinamarca   5.027000 -73.969000    164.0  4.000000e+08
3062         house     Atlántico  11.007000 -74.805000    114.0  3.100000e+08
3063         house    Bogotá D.C   4.685000 -74.148000    121.0  4.800000e+08
3064         house    Bogotá D.C   4.491000 -74.116000    150.0  9.900000e+08
3065     apartment  Cundinamarca   4.683932 -74.056925     82.0  5.900000e+08

          price_m2
0     1.702128e+06
1     6.538462e+06
3     4.046243e+06
4     2.457143e+06
5     5.050505e+06
…              …
3061  2.439024e+06
3062  2.719298e+06
3063  3.966942e+06
3064  6.600000e+06
3065  7.195122e+06

[2957 rows x 7 columns]
```

[43]:
```
df = df2
df2["price_cop"] = df.price_cop.astype(object)
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 2957 entries, 0 to 3065
Data columns (total 7 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   property_type  2957 non-null   object
 1   department     2957 non-null   object
 2   lat            2957 non-null   float64
 3   lon            2957 non-null   float64
 4   area_m2        2957 non-null   float64
 5   price_cop      2957 non-null   object
 6   price_m2       2957 non-null   float64
dtypes: float64(4), object(3)
memory usage: 184.8+ KB
```

6.6 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:

[44]:
```
df["department"].str[0:5:2]
```

[44]:
```
0       Mga
1       Bgt
3       Aln
4       Cni
5       Cni
       …
3061    Cni
3062    Aln
3063    Bgt
3064    Bgt
3065    Cni
Name: department, Length: 2957, dtype: object
```

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:

[45]:
```
df2['property_type'].str[0:5:1]
```

[45]:
```
0       house
1       house
3       house
4       house
5       house
       …
3061    house
3062    house
3063    house
3064    house
3065    apart
Name: property_type, Length: 2957, dtype: object
```

6.7 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:

[46]:
```
df["property_type"] = df["property_type"].str.replace("house", "single_family")
df.head()
```

[46]:

	property_type	department	lat	lon	area_m2	price_cop
0	single_family	Magdalena	11.233	-74.204	235.0	400000000.0
1	single_family	Bogotá D.C	4.714	-74.030	130.0	850000000.0

```
3  single_family     Atlántico  11.006 -74.808    346.0  1400000000.0
4  single_family  Cundinamarca   4.857 -74.061    175.0   430000000.0
5  single_family  Cundinamarca   4.883 -74.035    297.0  1500000000.0

       price_m2
0  1.702128e+06
1  6.538462e+06
3  4.046243e+06
4  2.457143e+06
5  5.050505e+06
```

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)`

[47]:
```
df["property_type"] = df["property_type"].str.replace("house", "single_family",␣
↪3)
df.head()
```

[47]:
```
   property_type    department     lat     lon  area_m2     price_cop  \
0  single_family     Magdalena  11.233 -74.204    235.0   400000000.0
1  single_family    Bogotá D.C   4.714 -74.030    130.0   850000000.0
3  single_family     Atlántico  11.006 -74.808    346.0  1400000000.0
4  single_family  Cundinamarca   4.857 -74.061    175.0   430000000.0
5  single_family  Cundinamarca   4.883 -74.035    297.0  1500000000.0

       price_m2
0  1.702128e+06
1  6.538462e+06
3  4.046243e+06
4  2.457143e+06
5  5.050505e+06
```

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.

[48]:
```
df = df2['property_type'].str.replace('apartment','multi_family',7)
df.head()
```

[48]:
```
0    single_family
1    single_family
3    single_family
4    single_family
5    single_family
```

```
Name: property_type, dtype: object
```

6.7.1 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:

[58]:
```
df.head()
```

[58]:
```
0    single_family
1    single_family
3    single_family
4    single_family
5    single_family
Name: property_type, dtype: object
```

[60]:
```
df.rename(columns={"property_type": "type_property"})
```

[60]:

	type_property	department	lat	lon	area_m2	price_usd
0	house	Bogotá D.C	4.690000	-74.048000	187.0	$330,899.98
1	house	Bogotá D.C	4.695000	-74.082000	82.0	$121,555.09
2	house	Quindío	4.535000	-75.676000	235.0	$219,474.47
3	house	Bogotá D.C	4.620000	-74.129000	195.0	$97,919.38
4	house	Atlántico	11.012000	-74.834000	112.0	$115,477.34
…	…	…	…	…	…	…
3061	house	Bogotá D.C	4.636000	-74.169000	227.0	$84,413.26
3062	house	Bolívar	10.384000	-75.474000	260.0	$303,887.73
3063	apartment	Cundinamarca	-0.001102	0.001431	87.0	$195,429.68
3064	apartment	Bogotá D.C	NaN	NaN	75.0	$114,802.03
3065	house	Bogotá D.C	4.721000	-74.068000	85.0	$131,684.68

[3066 rows x 6 columns]

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.

[55]:
```
df2.rename(columns={'lat':'latitude'})
```

[55]:

	property_type	department	latitude	lon	area_m2	\
0	single_family	Magdalena	11.233000	-74.204000	235.0	
1	single_family	Bogotá D.C	4.714000	-74.030000	130.0	
3	single_family	Atlántico	11.006000	-74.808000	346.0	
4	single_family	Cundinamarca	4.857000	-74.061000	175.0	
5	single_family	Cundinamarca	4.883000	-74.035000	297.0	

```
…              …            …          …         …       …
3061  single_family  Cundinamarca   5.027000 -73.969000   164.0
3062  single_family     Atlántico  11.007000 -74.805000   114.0
3063  single_family    Bogotá D.C   4.685000 -74.148000   121.0
3064  single_family    Bogotá D.C   4.491000 -74.116000   150.0
3065      apartment  Cundinamarca   4.683932 -74.056925    82.0

         price_cop      price_m2
0      400000000.0  1.702128e+06
1      850000000.0  6.538462e+06
3     1400000000.0  4.046243e+06
4      430000000.0  2.457143e+06
5     1500000000.0  5.050505e+06
…              …            …
3061   400000000.0  2.439024e+06
3062   310000000.0  2.719298e+06
3063   479999999.0  3.966942e+06
3064   990000000.0  6.600000e+06
3065   590000000.0  7.195122e+06

[2957 rows x 7 columns]
```

6.7.2 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:

[61]:
```
df["property_type"].unique()
```

[61]:
```
array(['house', 'apartment'], dtype=object)
```

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`:

[62]:
```
df2['department'].unique()
```

[62]:
```
array(['Magdalena', 'Bogotá D.C', 'Atlántico', 'Cundinamarca',
       'Valle del Cauca', 'Antioquia', 'Bolívar', 'Quindío', 'Caldas',
       'Boyacá', 'Meta', 'Santander',
       'San Andrés Providencia y Santa Catalina', 'Tolima', 'Risaralda',
       'Cesar'], dtype=object)
```

7 Concatenating

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

7.1 Concatenating DataFrames

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

[63]:
```
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)
```

```
df1 shape: (3066, 6)
df2 shape: (3066, 6)
```

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

[64]:
```
concat_df = pd.concat([df1, df2])
print("concat_df shape:", concat_df.shape)
concat_df.head()
```

```
concat_df shape: (6132, 7)
```

[64]:

	property_type	department	lat	lon	area_m2	price_usd	price_cop
0	house	Bogotá D.C	4.690	-74.048	187.0	$330,899.98	NaN
1	house	Bogotá D.C	4.695	-74.082	82.0	$121,555.09	NaN
2	house	Quindío	4.535	-75.676	235.0	$219,474.47	NaN
3	house	Bogotá D.C	4.620	-74.129	195.0	$97,919.38	NaN
4	house	Atlántico	11.012	-74.834	112.0	$115,477.34	NaN

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`.

[67]:
```
df2 =pd.read_csv("data/colombia-real-estate-2.csv")
df3 = pd.read_csv("data/colombia-real-estate-2.csv")
concat_df = pd.concat([df2,df3])
concat_df.head()
```

[67]:

	property_type	department	lat	lon	area_m2	price_cop
0	house	Magdalena	11.233	-74.204	235.0	4.000000e+08
1	house	Bogotá D.C	4.714	-74.030	130.0	8.500000e+08
2	house	Cundinamarca	4.851	-74.059	137.0	4.750000e+08
3	house	Atlántico	11.006	-74.808	346.0	1.400000e+09
4	house	Cundinamarca	4.857	-74.061	175.0	4.300000e+08

7.2 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.

[68]:
```
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())
```

```
len sr1: 3066
0    house
1    house
2    house
3    house
4    house
Name: property_type, dtype: object

len sr2: 3066
0    house
1    house
2    house
3    house
4    house
Name: property_type, dtype: object
```

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

[69]:
```
concat_sr = pd.concat([sr1, sr2])
print("len concat_sr:", len(concat_sr)),
print(concat_sr.head())
```

```
len concat_sr: 6132
0    house
1    house
2    house
3    house
4    house
Name: property_type, dtype: object
```

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.

[70]:
```
df1 = pd.read_csv("data/colombia-real-estate-2.csv")
df2=pd.read_csv("data/colombia-real-estate-3.csv")
s1=df1['area_m2']
s2=df2['area_m2']
result=pd.concat([s1,s2])
print(result)
```

```
0       235.0
1       130.0
2       137.0
3       346.0
4       175.0
        …
3060    200.0
3061     66.0
3062    330.0
3063    223.0
3064    262.0
Name: area_m2, Length: 6131, dtype: float64
```

8 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:

[ ]:
```
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.

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

9 References & Further Reading

- Tutorial for `shape`
- Tutorial for `info`
- Adding columns to a DataFrame
- Creating DataFrame from dictionary
- Working with JSON
- Dropping columns from a DataFrame
- Splitting columns in a DataFrame

- Recasting values
- Replacing strings
- Concatenating DataFrames
- From DataFrames to Series
- Stack Overflow: What is serialization
- Understand Python Pickling
