Pandas: Advanced

Calculate Summary Statistics for a DataFrame or Series

Many datasets are large, and it can be helpful to get a summary of information in them. Let's load a dataset and examine the first few rows:

In [ ]:
```
import pandas as pd

mexico_city1 = pd.read_csv("./data/mexico-city-real-estate-1.csv")
mexico_city1.head
```

Let's get a summary description of this dataset.

In [ ]:
```
mexico_city1.describe()
```

Like most large datasets, this one has many missing values which are missing. The describe function will ignore missing values in each column. You can also remove rows and columns with missing values, and then get a summary of the data that's still there. We need to remove columns first, before removing the rows; the sequence of operations here is important. The code looks like this:

In [ ]:
```
mexico_city1 = mexico_city1.drop(
    ["floor", "price_usd_per_m2", "expenses", "rooms"], axis=1
)
mexico_city1 = mexico_city1.dropna(axis=0)
mexico_city1.head
```

Let's take a look at our new, cleaner dataset.

In [ ]:
```
mexico_city1.describe()
```

Practice

Reload the `mexico-city-real-estate-1.csv` dataset. Reverse the sequence of operations by first dropping all rows where there is a missing value, and then dropping the columns, `floor`, `price_usd_per_m2`, `expenses` and `rooms`. What is the size of the resulting DataFrame?

In [ ]:
```
mexico_city1 = pd.read_csv("./data/mexico-city-real-estate-1.csv")
mexico_city1 = ...
mexico_city1 = mexico_city1.drop(
    ["floor", "price_usd_per_m2", "expenses", "rooms"], axis=1
)  # REMOVERHS
print(mexico_city1.shape)
```

Select a Series from a DataFrame

Since the datasets we work with are so large, you might want to focus on a single column of a DataFrame. Let's load up the `mexico-city-real-estate-2` dataset, and examine the first few rows to find the column names.

In [ ]:
```
mexico_city2 = pd.read_csv("./data/mexico-city-real-estate-2.csv")
mexico_city2.head
```

Maybe we're interested in the `surface_covered_in_m2` column. The code to extract just that one column looks like this:

In [ ]:
```
surface_covered_in_m2 = mexico_city2["surface_covered_in_m2"]
surface_covered_in_m2
```

Practice

Select the `price` series from the `mexico-city-real-estate-2` dataset, and load it into the `mexico_city2` DataFrame

In [ ]:
```
price = ...
print(price)
```

Subset a DataFrame by Selecting One or More Columns

You may find it more efficient to work with a smaller portion of a dataset that's relevant to you. One way to do this is to select some columns from a DataFrame and make a new DataFrame with them. Let's load a dataset to do this and examine the first few rows to find the column headings:

In [ ]:
```
mexico_city4 = pd.read_csv("./data/mexico-city-real-estate-4.csv")
mexico_city4.head
```

Let's choose `"operation"`, `"property_type"`, `"place_with_parent_names"`, and `"price"`:

In [ ]:
```
mexico_city4_subset = mexico_city4[
    ["operation", "property_type", "place_with_parent_names", "price"]
]
```

Once we've done that, we can find the resulting number of entries in the DataFrame:

In [ ]:
```
mexico_city4_subset.shape
```

Practice

Load the `mexico-city-real-estate-1.csv` dataset and subset it to obtain the `operation`, `lat-lon` and `place_with_property_names` columns only. How many entries are in the resulting DataFrame?

In [ ]:
```
mexico_city1 = ...
mexico_city1_subset = mexico_city1[
    ["operation", "lat-lon", "place_with_parent_names"]
]  # REMOVERHS
print(mexico_city1_subset.shape)
```

Subset the Columns of a DataFrame Based on Data Types

It's helpful to be able to find specific types of entries — typically numeric ones — and put all of these in a separate DataFrame. First, let's take a look at the `mexico-city-real-estate-5` dataset.

In [ ]:
```
mexico_city5 = pd.read_csv("./data/mexico-city-real-estate-5.csv")
mexico_city5.head
```

The code to subset just the numerical entries looks like this:

In [ ]:
```
mexico_city5_numbers = mexico_city5.select_dtypes(include="number")
mexico_city5_numbers.head
```

Practice

Create a subset of the DataFrame from `mexico-city-real-estate-5` which excludes numbers.

In [ ]:
```
mexico_city3 = ...
mexico_city3_no_numbers = ...
print(mexico_city3_no_numbers.shape)
```

Working with value_counts in a Series

In order to use the data in a series for other types of analysis, it might be helpful to know how often each value occurs in the Series. To do that, we use the `value_counts` method to aggregate the data. Let's take a look at the number of properties associated with each department in the `colombia-real-estate-1` dataset.

In [ ]:
```
df1 = pd.read_csv("data/colombia-real-estate-1.csv", usecols=["department"])
df1["department"].value_counts()
```

Practice

Try it yourself! Aggregate the different property types in the `colombia-real-estate-2` dataset.

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

Series and `Groupby`

Large Series often include data points that have some attribute in common, but which are nevertheless not grouped together in the dataset. Happily, pandas has a method that will bring these data points together into groups.

Let's take a look at the `colombia-real-estate-1` dataset. The set includes properties scattered across Colombia, so it might be useful to group properties from the same department together; to do this, we'll use the `groupby` method. The code looks like this:

In [ ]:
```
dept_group = df1.groupby("department")
```

To make sure we got all the departments in the dataset, let's print the first occurrence of each department.

In [ ]:
```
dept_group.first()
```

Practice

Try it yourself! Group the properties in `colombia-real-estate-2` by department, and print the result.

In [ ]:
```
df2 = ...
dept_group = ...
dept_group.first()
```

Now that we have all the properties grouped by department, we might want to see the properties in just one of the departments. We can use the `get_group` method to do that. If we just wanted to see the properties in `"Santander"`, for example, the code would look like this:

In [ ]:
```
dept_group1 = df1.groupby("department")
dept_group1.get_group("Santander")
```

We can also make groups based on more than one category by adding them to the `groupby` method. After resetting the `df1` DataFrame, here's what the code looks like if we want to group properties both by `department` and by `property_type`.

In [ ]:
```
df1 = pd.read_csv("data/colombia-real-estate-1.csv")
dept_group2 = df1.groupby(["department", "property_type"])
dept_group2.first()
```

Practice

Try it yourself! Group the properties in `colombia-real-estate-2` by department and property type, and print the result.

In [ ]:
```
dept_group = ...
dept_group.first()
```

Finally, it's possible to use `groupby` to calculate aggregations. For example, if we wanted to find the average property area in each department, we would use the `.mean()` method. This is what the code for that looks like:

In [ ]:
```
dept_group = df1.groupby("department")["area_m2"].mean()
dept_group
```

Practice Try it yourself! Use the `.mean` method in the `colombia-real-estate-2` dataset to find the average price in Colombian pesos (`"price_cop"`) for properties in each `"department"` .

In [ ]:
```
dept_group = ...
dept_group
```

Subset a DataFrame's Columns Based on the Column Data Types

It's helpful to be able to find entries of a certain type, typically numerical entries, and put all of these in a separate DataFrame. Let's load a dataset to see how this works:

In [ ]:
```
mexico_city5 = pd.read_csv("./data/mexico-city-real-estate-5.csv")
mexico_city5.head
```

Now let's get only numerical entries:

In [ ]:
```
mexico_city5_numbers = mexico_city5.select_dtypes(include="number")
mexico_city5_numbers.head
```

Let's now find all entries which are not numerical entries:

In [ ]:
```
mexico_city5_no_numbers = mexico_city5.select_dtypes(exclude="number")
mexico_city5_no_numbers.head
```

Practice

Create a subset of the DataFrame from `mexico-city-real-estate-5.csv` which excludes numbers. How many entries does it have?

In [ ]:
```
mexico_city3 = ...
```

```
mexico_city3_no_numbers = ...
print(mexico_city3_no_numbers.shape)
```

Subset a DataFrame's columns based on column names

To subset a DataFrame by column names, you can either define a list of columns include or define a list of columns to exclude. Next, retain or drop the columns accordingly. For example, let's suppose we want to modify the `mexico_city3` dataset and only retain the first three columns. Let's define two lists, one with the columns to retain and one with the columns to drop:

In [ ]:
```
drop_cols = [
    "lat-lon",
    "price",
    "currency",
    "price_aprox_local_currency",
    "price_aprox_usd",
    "surface_total_in_m2",
    "surface_covered_in_m2",
    "price_usd_per_m2",
    "price_per_m2",
    "floor",
    "rooms",
    "expenses",
    "properati_url",
]

keep_cols = ["operation", "property_type", "place_with_parent_names"]
```

Next, we'll explore both approaches to subset `mexico_city3`. To retain columns based on `keep_cols`:

In [ ]:
```
mexico_city3_subsetted = mexico_city3[keep_cols]
```

To drop columns in `drop_cols`:

In [ ]:
```
mexico_city3_subsetted = mexico_city3.drop(columns=drop_cols)
```

Practice

Create a subset of the DataFrame from `mexico-city-real-estate-3.csv` which excludes the last two columns.

In [ ]:
```

```

Pivot Tables

A pivot table allows to aggregate and summarize a DataFrame across multiple variables. For example, let's suppose we wanted to calculate the mean of the `price` column in the `mexico_city3` dataset for the different values in the `property_type` column:

In [ ]:
```
import numpy as np

mexico_city3_pivot = ...
mexico_city3_pivot
```

Subsetting with Masks

Another way to to create subsets from a larger dataset is through **masking**. Masks are ways to filter out the data you're not interested in so that you can focus on the data that you are. For example, we might want to look at properties in Colombia that are bigger than 200 square meters. In order to create this subset, we'll need to use a mask.

First, we'll reset our `df1` DataFrame so that we can draw on all the data in its original form. Then we'll create a statement and then assign the result to `mask`.

In [ ]:
```
df1 = pd.read_csv("data/colombia-real-estate-1.csv")
mask = df1["area_m2"] > 200
mask.head()
```

Notice that `mask` is a Series of Boolean values. Where properties are smaller than 200 square meters, our statement evaluates as `False`; where they're bigger than 200, it evaluates to `True`.

Once we have our mask, we can use it to select all the rows from `df1` that evaluate as `True`.

In [ ]:
```
df1[mask].head()
```

Practice

Try it yourself! Read `colombia-real-estate-2` into a DataFrame named `df2`, and create a mask to select all the properties that are smaller than 100 square meters.

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

We can also create masks with multiple criteria using `&` for "and" and `|` for "or." For example, here's how we would find all properties in Atlántico that are bigger than 400 square meters.

In [ ]:
```
mask = (df1["area_m2"] > 400) & (df1["department"] == "Atlántico")
df1[mask].head()
```

Practice

Try it yourself! Create a mask for `df2` to select all the properties in Tolima that are smaller than 150 square meters.

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

Reshape a DataFrame based on column values

What's a pivot table?

A pivot table allows you to quickly aggregate and summarize a DataFrame using an aggregation function. For example, to build a pivot table that summarizes the mean of the `price_cop` column for each of the unique categories in the `property_type` column in `df2`:

In [ ]:
```
import numpy as np

pivot1 = pd.pivot_table(df2, values="price_cop", index="property_type", aggfunc=np.mean
pivot1
```

Practice

Try it yourself! build a pivot table that summarizes the mean of the `price_cop` column for each of the unique departments in the `department` column in `df2`:

In [ ]:
```
# REMOVE {
pivot2 = pd.pivot_table(df2, values="price_cop", index="department", aggfunc=np.mean)
# REMOVE }
pivot2
```

Combine multiple categories in a Series

Categorical variables can be collapsed into a fewer number of categories. One approach is to retain the values of the most frequently observed values and collapse all remaining values into a single category. For example, to retain only the values of the top 10 most frequent categories in the `department` column and then collapse the other categories together, use `value_counts` to generate the count of the values:

In [ ]:
```
df2["department"].value_counts()
```

Next, select just the top 10 using the `head()` method, and select the column names by using the `index` attribute of the series:

In [ ]:
```
top_10 = df2["department"].value_counts().head(10).index
```

Finally, use the apply method and a lambda function to select only the values from the `department` column and collapse the remaining values into the value `Other`:

In [ ]:
```
df2["department"] = df2["department"].apply(lambda x: x if x in top_10 else "Other")
```

Practice

Try it yourself! Retain the remaining top 5 most frequent values in the `department` column and collapse the remaining values into the category `Other`.

In [ ]:
```

```

References & Further Reading

- Pandas Documentation on Dropping a Column in a DataFrame
- Pandas Documentation on Printing out the First Few Rows of a DataFrame
- Pandas Documentation on Reading in a CSV File Into a DataFrame
- Getting Started with Pandas Documentation
- Pandas Documentation on Extracting a Column to a Series
- Pandas Official Indexing Guide
- Series in pandas
- Tutorial for `value_counts`
- Tutorial for `groupby`
- pandas Documentation for `groupby`
- Pandas Official Indexing Guide
- Online Examples of Selecting Numeric Columns of a DataFrame
- Official Pandas Documentation on Data Types in a DataFrame
- Pandas documentation for Boolean indexing
- More information on creating various kinds of subsets
- More on boolean indexing
- A tutorial on using various criteria to create subsets
