5 Pivoting and Joining

5.1 Introduction

This document demonstrates the use of the `pandas` library in Python to do pivoting and joining of datasets.

Note

If you do not have the `pandas` library installed then you will need to run

`pip install pandas`

in the Jupyter terminal to install. **Remember:** you only need to install once per machine (or Colab session).

```
import pandas as pd
```

```
# Population data from GapMinder
population = pd.read_csv("/content/pop.csv")
```

5.2 Pivoting Data in Python

Data come in all shapes and forms! Rare is the day when we can open a dataset for the first time and it's ready for every type of visualization or analysis that we could want to do with it.

In addition to the wrangling we discussed in the previous chapter, there may be a need to reshape the dataset entirely. For example, the column names might be values themselves that we want to make use of.

Recall our introduction of *tidy* data in the previous chapter…

5.2.1 Tidy Data is Special Tabular Data

For most people, the image that comes to mind when thinking about data is indeed something tabular or spreadsheet-like in nature. **Which is great!**

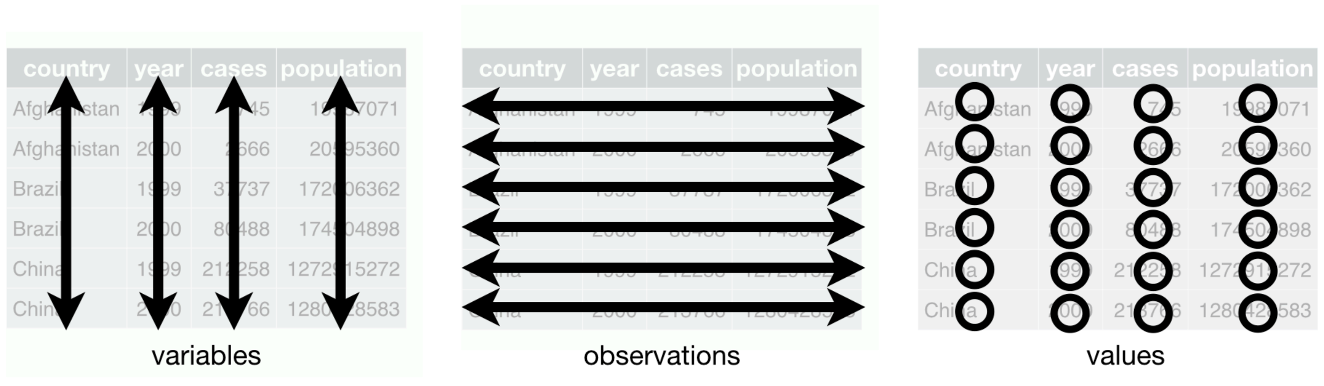
Tabular data is a form preferred by MANY different data operations and work. However, we will want to take this one step further. In almost all data science work we want our data to be **tidy**

Note

A dataset is **tidy** if it adheres to following three characteristics:

- Every column is a variable

- Every row is an observation
- Every cell is a single value



In the previous chapter you were asked to open up a GapMinder dataset here and to comment on whether this dataset was *tidy* or not. The answer was **no**, this dataset is not *tidy*. These datasets come with a row representing a country, each column representing a year, and each cell representing the value of the global indicator selected. To be *tidy* these three variables (`country`, `year`, `global indicator`) should each have their own column, instead of the `year` variable taking values as the column headers.

5.2.2 Wide to Long Format

The GapMinder dataset is an example of what's commonly referred to as data in a *wide format*. To make this dataset *tidy* we aim for a dataset with columns for `country`, `year`, and `global indicator` (e.g. population). Three columns is many fewer than the current number of columns, and so we will convert this dataset from *wide to long format*.

⚠ **Warning**

It often helps to physically draw/map out what our current dataset looks like and what the look of our target dataset is, before actually trying to write any code to do this. Writing the code can be **extremely** easier after this exercise, and only makes future pivot operations easier.

In order to convert our dataset from *wide to long format* we will use `.melt()` (or `.wide_to_long()`) in `pandas`.

```
long_population = population.melt(id_vars=["country"], var_name="year", value_nam
```

Check In

With 2-3 people around you navigate to GapMinder, download the population dataset, and convert it from wide to long format. Does the result look how you expect? Is any other wrangling necessary?

5.2.3 Long to Wide Format

Even though certain data shapes are not considered *tidy*, they may be more conducive to performing certain operations than other shapes. For example, what if we were interested in the change in country population between 1950 and 2010? In the original *wide* shape of the GapMinder data this operation would have been a simple difference of columns like below.

```
population["pop_diff"] = population["2010"] - population["1950"]
```

Check In

Why doesn't the above code work without further wrangling? What in the dataset needs to change for this operation to work?

In the *long* format of our Gapminder dataset (`long_population`), this operation is less straightforward. Sometimes datasets come to us in *long* format and to do things like the operation above we need to convert that dataset from *long to wide format*. We can go the reverse direction (i.e. *long to wide format*) with `.pivot()` in `pandas`.

```
wide_population = long_population.pivot(index = "country", columns = "year", valu
wide_population = wide_population.reset_index()
```

Learn More

We haven't spent much time discussing the `index` of a `pandas` DataFrame, but you can think of it like an address for data, or slices of data in a DataFrame. You can also think of an `index` (or `indices`) as row names, or axis labels, for your dataset. This can be useful for a number of functions in Python, and can enhance the look of results or visualizations.

However, understanding them is not critical for what we will do in Python. Furthermore, variables that are `indices` for a DataFrame cannot be accessed or referenced in the same way as other variables in the DataFrame. So, we will avoid their use if possible.

5.3 Joining Datasets in Python

The information you need is often spread across multiple data sets, so you will need to combine multiple data sets into one. In this chapter, we discuss strategies for combining information from multiple (tabular) data sets.

As a working example, we will use a data set of baby names collected by the Social Security Administration. Each data set in this collection contains the names of all babies born in the United States in a particular year. This data is publicly available, and a copy has been made available at https://dlsun.github.io/pods/data/names/.

5.3.1 Concatenating and Merging Data

5.3.1.1 Concatenation

Sometimes, the *rows* of data are spread across multiple files, and we want to combine the rows into a single data set. The process of combining rows from different data sets is known as concatenation.

Visually, to concatenate two `DataFrames`, we simply stack them on top of one another.

For example, suppose we want to understand how the popularity of different names evolved between 1995 and 2015. The 1995 names and the 2015 names are stored in two different files: `yob1995.txt` and `yob2015.txt`, respectively. To carry out this analysis, we will need to combine these two data sets into one.

```
data_dir = "http://dlsun.github.io/pods/data/names/"
names1995 = pd.read_csv(data_dir + "yob1995.txt",
                        header=None,
                        names=["Name", "Sex", "Count"])
names1995
```

	Name	Sex	Count
0	Jessica	F	27935
1	Ashley	F	26603
2	Emily	F	24378
3	Samantha	F	21646
4	Sarah	F	21369
...	...	...	...
26075	Zerek	M	5
26076	Zhen	M	5
26077	Ziggy	M	5
26078	Zuberi	M	5
26079	Zyon	M	5

26080 rows × 3 columns

```
names2015 = pd.read_csv(data_dir + "yob2015.txt",
                        header=None,
                        names=["Name", "Sex", "Count"])
names2015
```

	Name	Sex	Count
0	Emma	F	20455
1	Olivia	F	19691
2	Sophia	F	17417
3	Ava	F	16378
4	Isabella	F	15617
...	...	...	...
33116	Zykell	M	5

	Name	Sex	Count
33117	Zyking	M	5
33118	Zykir	M	5
33119	Zyrus	M	5
33120	Zyus	M	5

33121 rows × 3 columns

To concatenate the two, we use the `pd.concat()` function, which accepts a *list* of `pandas` objects (`DataFrames` or `Series`) and concatenates them.

```
pd.concat([names1995, names2015])
```

	Name	Sex	Count
0	Jessica	F	27935
1	Ashley	F	26603
2	Emily	F	24378
3	Samantha	F	21646
4	Sarah	F	21369
...	...	...	...
33116	Zykell	M	5
33117	Zyking	M	5
33118	Zykir	M	5
33119	Zyrus	M	5
33120	Zyus	M	5

59201 rows × 3 columns

1. There is no longer any way to distinguish the 1995 data from the 2015 data. To fix this, we can add a **Year** column to each `DataFrame` before we concatenate.

2. The indexes from the original `DataFrames` are preserved in the concatenated `DataFrame`. (To see this, observe that the last index in the `DataFrame` is about 33000, which corresponds to the number of rows in `names2015`, even though there are 59000 rows in the `DataFrame`.) That means that there are two rows with an index of 0, two rows with an index of 1, and so on. To force `pandas` to generate a completely new index for this `DataFrame`, ignoring the indices from the original `DataFrames`, we specify `ignore_index=True`.

```
names1995["Year"] = 1995
names2015["Year"] = 2015
names = pd.concat([names1995, names2015], ignore_index=True)
names
```

	Name	Sex	Count	Year
0	Jessica	F	27935	1995
1	Ashley	F	26603	1995
2	Emily	F	24378	1995
3	Samantha	F	21646	1995
4	Sarah	F	21369	1995
...	...	...	...	...
59196	Zykell	M	5	2015
59197	Zyking	M	5	2015
59198	Zykir	M	5	2015
59199	Zyrus	M	5	2015
59200	Zyus	M	5	2015

59201 rows × 4 columns

Now this is a `DataFrame` we can use!

5.3.1.2 Merging (a.k.a Joining)

More commonly, the data sets that we want to combine actually contain different information about the same observations. In other words, instead of stacking the `DataFrames` on top of each other, as in concatenation, we want to stack them next to each other. The process of combining columns or variables from different data sets is known as *merging* or *joining*.

The observations may be in a different order in the two data sets, so merging is not as simple as placing the two `DataFrames` side-by-side.

Merging is an operation on two `DataFrames` that returns a third `DataFrame`. By convention, the first `DataFrame` is referred to as the one on the "left", while the second `DataFrame` is the one on the "right".

This naming convention is reflected in the syntax of the `.merge()` function in `pandas`. In the code below, the "left" `DataFrame`, `names1995`, is quite literally on the left in the code, while the "right" `DataFrame`, `names2015`, is to the right. We also specify the variables to match across the two `DataFrames`.

```
names1995.merge(names2015, on=["Name", "Sex"])
```

	Name	Sex	Count_x	Year_x	Count_y	Year_y
0	Jessica	F	27935	1995	1587	2015
1	Ashley	F	26603	1995	3424	2015
2	Emily	F	24378	1995	11786	2015
3	Samantha	F	21646	1995	5340	2015
4	Sarah	F	21369	1995	4521	2015

	Name	Sex	Count_x	Year_x	Count_y	Year_y
...	...	...	...	...	...	...
15675	Zephan	M	5	1995	23	2015
15676	Zeppelin	M	5	1995	70	2015
15677	Zerek	M	5	1995	5	2015
15678	Ziggy	M	5	1995	44	2015
15679	Zyon	M	5	1995	148	2015

15680 rows × 6 columns

The **most important component** of *merging* two datasets is the presence of at least one *key* variable that both datasets share. This variable is sometimes referred to as an *ID variable*. It's this variable that we will want to *merge on*, i.e. use to combine the two datasets intelligently.

The variables that we joined on (`Name` and `Sex`) appear once in the final `DataFrame`. The variable `Count`, which we did not join on, appears twice—since there was a column called `Count` in both of the original `DataFrames`. Notice that `pandas` automatically appended the suffix `_x` to the name of the variable from the left `DataFrame` and `_y` to the one from the right `DataFrame`. We can customize the suffixes by specifying the `suffixes=` parameter.

```
names1995.merge(names2015, on=["Name", "Sex"], suffixes=("1995", "2015"))
```

	Name	Sex	Count1995	Year1995	Count2015	Year2015
0	Jessica	F	27935	1995	1587	2015
1	Ashley	F	26603	1995	3424	2015
2	Emily	F	24378	1995	11786	2015
3	Samantha	F	21646	1995	5340	2015
4	Sarah	F	21369	1995	4521	2015
...	...	...	...	...	...	...
15675	Zephan	M	5	1995	23	2015
15676	Zeppelin	M	5	1995	70	2015
15677	Zerek	M	5	1995	5	2015
15678	Ziggy	M	5	1995	44	2015
15679	Zyon	M	5	1995	148	2015

15680 rows × 6 columns

In the code above, we assumed that the columns that we joined on had the same names in the two data sets. What if they had different names? For example, suppose the variable had been called `Sex` in one data set and `Gender` in the other. We can specify which variables to use from the left and right data sets using the `left_on=` and `right_on=` parameters.

```
# Create new DataFrames where the column names are different
names2015_ = names2015.rename({"Sex": "Gender"}, axis=1)

# This is how you merge them.
names1995.merge(
    names2015_,
    left_on=("Name", "Sex"),
    right_on=("Name", "Gender")
)
```

	Name	Sex	Count_x	Year_x	Gender	Count_y	Year_y
0	Jessica	F	27935	1995	F	1587	2015
1	Ashley	F	26603	1995	F	3424	2015
2	Emily	F	24378	1995	F	11786	2015
3	Samantha	F	21646	1995	F	5340	2015
4	Sarah	F	21369	1995	F	4521	2015
...	...	...	...	...	...	...	...
15675	Zephan	M	5	1995	M	23	2015
15676	Zeppelin	M	5	1995	M	70	2015
15677	Zerek	M	5	1995	M	5	2015
15678	Ziggy	M	5	1995	M	44	2015
15679	Zyon	M	5	1995	M	148	2015

15680 rows × 7 columns

5.3.1.3 One-to-One and Many-to-One Relationships

In the example above, there was at most one combination of `Name` and `Sex` in the 2015 data set for each combination of `Name` and `Sex` in the 1995 data set. These two data sets are thus said to have a one-to-one relationship. The same would be true of combining two GapMinder datasets.

However, two data sets need not have a one-to-one relationship! Two datasets could have a *many-to-one relationship*. In general, it's **extremely important** to think carefully about what variables each of your two datasets have to begin with, and what variables you want your merged dataset to have…and what that merged dataset will represent with respect to your data.

5.3.1.4 Many-to-Many Relationships: A Cautionary Tale

It is also possible for multiple rows in the left `DataFrame` to match multiple rows in the right `DataFrame`. In this case, the two data sets are said to have a *many-to-many relationship*. Many-to-many joins can lead to misleading analyses, so it is important to exercise caution when working with many-to-many relationships.

For example, in the baby names data set, the `Name` variable is not uniquely identifying. For example, there are both males and females with the name "Jessie".

```python
jessie1995 = names1995[names1995["Name"] == "Jessie"]
jessie1995
```

	Name	Sex	Count	Year
248	Jessie	F	1138	1995
16047	Jessie	M	903	1995

```python
jessie2015 = names2015[names2015["Name"] == "Jessie"]
jessie2015
```

	Name	Sex	Count	Year
615	Jessie	F	469	2015
20009	Jessie	M	233	2015

If we join these two `DataFrames` on `Name`, then we will end up with a many-to-many join, since each "Jessie" row in the 1995 data will be paired with each "Jessie" row in the 2015 data.

```python
jessie1995.merge(jessie2015, on=["Name"])
```

	Name	Sex_x	Count_x	Year_x	Sex_y	Count_y	Year_y
0	Jessie	F	1138	1995	F	469	2015
1	Jessie	F	1138	1995	M	233	2015
2	Jessie	M	903	1995	F	469	2015
3	Jessie	M	903	1995	M	233	2015

Notice that Jessie ends up appearing four times:

- Female Jessies from 1995 are matched with female Jessies from 2015. (Good!)
- Male Jessies from 1995 are matched with male Jessies from 2015. (Good!)
- Female Jessies from 1995 are matched with male Jessies from 2015. (This is perhaps undesirable.)
- Male Jessies from 1995 are matched with female Jessies from 2015. (Also unexpected and undesirable.)

If we had used a data set like this to determine the number of Jessies in 1995, then we would end up with the wrong answer, since we would have double-counted both female and male Jessies as a result of the many-to-many join. This is why it is important to exercise caution when working with (potential) many-to-many relationships.

5.3.2 Types of Joins

Above, we saw how to merge (or join) two data sets by matching on certain variables. But what happens when a row in one `DataFrame` has no match in the other?

First, let's investigate how `pandas` handles this situation by default. The name "Nevaeh", which is "heaven" spelled backwards, took after Sonny Sandoval of the band P.O.D. gave his daughter the name in 2000. Let's look at how common this name was five years earlier and five years after.

```
data_dir = "http://dlsun.github.io/pods/data/names/"

names1995 = pd.read_csv(data_dir + "yob1995.txt",
                        header=None, names=["Name", "Sex", "Count"])
names2005 = pd.read_csv(data_dir + "yob2005.txt",
                        header=None, names=["Name", "Sex", "Count"])
```

```
names1995[names1995.Name == "Nevaeh"]
```

Name	Sex	Count

```
names2005[names2005.Name == "Nevaeh"]
```

	Name	Sex	Count
68	Nevaeh	F	4552
21353	Nevaeh	M	56

In 1995, there were no girls (at least fewer than 5) named Nevaeh; just eight years later, there were over 4500 girls (and even 56 boys) with the name. It seems like Sonny Sandoval had a huge effect.

What happens to the name "Nevaeh" when we merge the two data sets?

```
names = names1995.merge(names2005, on=["Name", "Sex"])
names[names.Name == "Nevaeh"]
```

Name	Sex	Count_x	Count_y

By default, `pandas` only includes combinations that are present in both `DataFrames`. If it cannot find a match for a row in one `DataFrame`, then the combination is simply dropped.

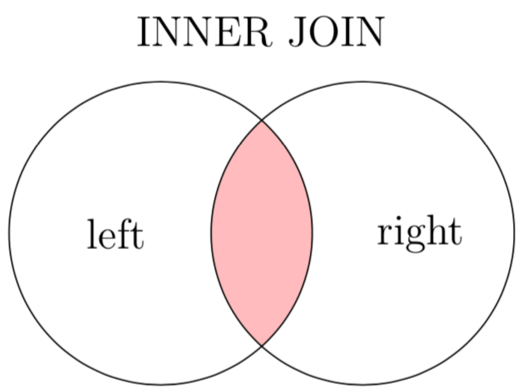
But in this context, the fact that a name does not appear in one data set is informative. It means that no babies were born in that year with that name. We might want to include names that appeared in only one of the two `DataFrames`, rather than just the names that appeared in both.

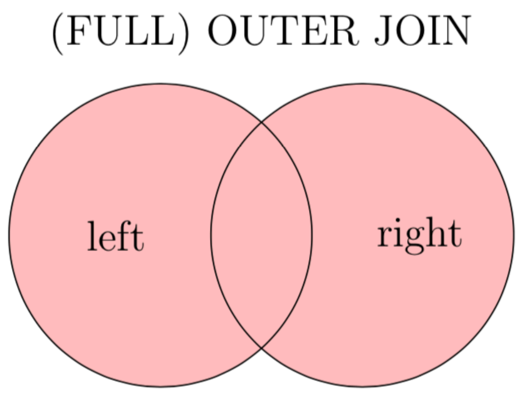
There are four types of joins, distinguished by whether they include the rows from the left `DataFrame`, the right `DataFrame`, both, or neither:

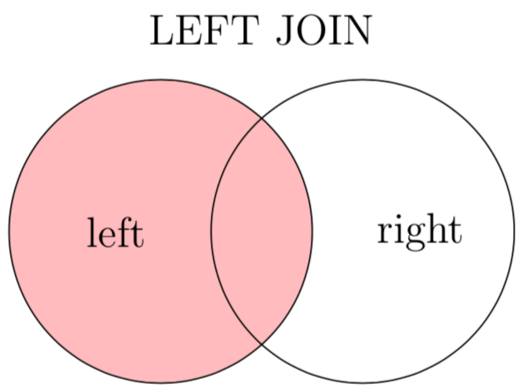
1. *inner join* (default): only values that are present in both `DataFrames` are included in the result

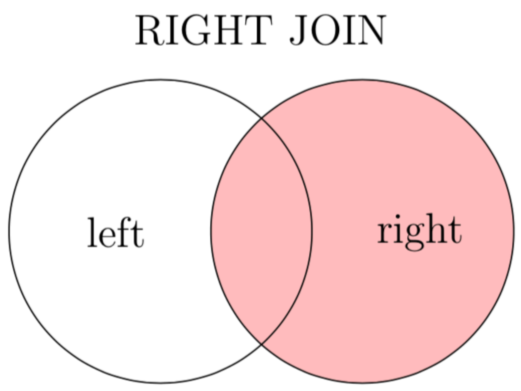
2. *outer join*: any value that appears in either `DataFrame` is included in the result
3. *left join*: any value that appears in the left `DataFrame` is included in the result, whether or not it appears in the right `DataFrame`
4. *right join*: any value that appears in the right `DataFrame` is included in the result, whether or not it appears in the left `DataFrame`.

One way to visualize the different types of joins is using Venn diagrams. The shaded region indicates which rows that are included in the output. For example, only rows that appear in both the left and right `DataFrames` are included in the output of an inner join.









In `pandas`, the join type is specified using the `how=` argument.

Now let's look at the examples of each of these types of joins.

```python
# inner join
names_inner = names1995.merge(names2005, on=["Name", "Sex"], how="inner")
names_inner
```

	Name	Sex	Count_x	Count_y
0	Jessica	F	27935	8108
1	Ashley	F	26603	13270
2	Emily	F	24378	23930

	Name	Sex	Count_x	Count_y
3	Samantha	F	21646	13633
4	Sarah	F	21369	11527
...	...	...	...	...
19119	Zeppelin	M	5	7
19120	Zerek	M	5	8
19121	Zhen	M	5	7
19122	Ziggy	M	5	6
19123	Zyon	M	5	102

19124 rows × 4 columns

```
# outer join
names_outer = names1995.merge(names2005, on=["Name", "Sex"], how="outer")
names_outer
```

	Name	Sex	Count_x	Count_y
0	Jessica	F	27935.0	8108.0
1	Ashley	F	26603.0	13270.0
2	Emily	F	24378.0	23930.0
3	Samantha	F	21646.0	13633.0
4	Sarah	F	21369.0	11527.0
...	...	...	...	...
39490	Zymiere	M	NaN	5.0
39491	Zyrell	M	NaN	5.0
39492	Zyrian	M	NaN	5.0
39493	Zyshon	M	NaN	5.0
39494	Zytavious	M	NaN	5.0

39495 rows × 4 columns

Names like “Zyrell” and “Zyron” appeared in the 2005 data but not the 1995 data. For this reason, their count in 1995 is `NaN`. In general, there will be missing values in `DataFrames` that result from an outer join. Any time a value appears in one `DataFrame` but not the other, there will be `NaN`s in the columns from the `DataFrame` missing that value.

```
names_inner.isnull().sum()
```

```
Name       0
Sex        0
Count_x    0
```

```
Count_y    0
dtype: int64
```

Left and right joins preserve data from one `DataFrame` but not the other. For example, if we were trying to calculate the percentage change for each name from 1995 to 2005, we would want to include all of the names that appeared in the 1995 data. If the name did not appear in the 2005 data, then that is informative.

```
# left join
names_left = names1995.merge(names2005, on=["Name", "Sex"], how="left")
names_left
```

	Name	Sex	Count_x	Count_y
0	Jessica	F	27935	8108.0
1	Ashley	F	26603	13270.0
2	Emily	F	24378	23930.0
3	Samantha	F	21646	13633.0
4	Sarah	F	21369	11527.0
...	...	...	...	...
26075	Zerek	M	5	8.0
26076	Zhen	M	5	7.0
26077	Ziggy	M	5	6.0
26078	Zuberi	M	5	NaN
26079	Zyon	M	5	102.0

26080 rows × 4 columns

The result of the left join has `NaN` s in the columns from the right `DataFrame`.

```
names_left.isnull().sum()
```

```
Name          0
Sex           0
Count_x       0
Count_y    6956
dtype: int64
```

The result of the right join, on the other hand, has `NaN` s in the column from the left `DataFrame`.

```
# right join
names_right = names1995.merge(names2005, on=["Name", "Sex"], how="right")
names_right
```

	Name	Sex	Count_x	Count_y
0	Emily	F	24378.0	23930
1	Emma	F	5041.0	20335
2	Madison	F	9775.0	19562
3	Abigail	F	7821.0	15747
4	Olivia	F	7624.0	15691
...	...	...	...	...
32534	Zymiere	M	NaN	5
32535	Zyrell	M	NaN	5
32536	Zyrian	M	NaN	5
32537	Zyshon	M	NaN	5
32538	Zytavious	M	NaN	5

32539 rows × 4 columns

```
names_right.isnull().sum()
```

```
Name           0
Sex            0
Count_x    13415
Count_y        0
dtype: int64
```

Check In

Download a second GapMinder dataset and merge it with the population dataset from above. Did you have to pivot first? Which order of operations makes the most sense? Is your resulting dataset *tidy*?

Practice Activity

Click here to solve a riddle using data manipulation.

5.4 Data Wrangling and AI

The advice in this section applies to data analysis tasks in general, not only to the basic wrangling and summarizing in this chapter - but since this is our first foray in to data wrangling code, let's dive right in!

In data processing and summarizing, we can think of four main ways that genAI can support your work. In this chat, we make use of all four of the steps below; open it up and follow along.

5.4.0.1 1. As a **planner**, to help you chart out your wrangling steps.

Data analysis involves going from Point A (your current dataset) to Point B (your desired calculation or visualization). One of the most difficult aspects of data science is figuring out what intermediate steps will get you there. For this, a genAI tool can be a great "thinking buddy" and suggest a path forward.

Although it may not give a perfect solution (especially since it typically cannot see your data), two heads are often better than one. The generated suggestions may include data cleaning steps you hadn't thought of, or help you fill in a gap when you are stuck.

As always, you will get your best results if your prompt is **well-specified:** make sure to describe the structure of the dataset you have, and the exact nature of the output you are trying to produce.

Example

In our chat example, we got back some pretty useful steps, including a great reminder to make sure our date-time data is the proper *type*. However, because we weren't overly detailed about the dataset, the AI had to "guess" about which columns existed. And since we also didn't clarify what we wanted our plot to look like, it had to offer an "optional" Step 6.

5.4.0.2 2. As **documentation**, when you are struggling with the exact use case or syntax of a function.

It is extremely common in coding to know what function(s) you need for your task, but to be a little fuzzy on the details of how to make then run the way you want. While the information you need can technically always be found in the official documentation of the functions, these can sometimes be difficult to understand. One of the absolute *strongest contributions* of genAI is the ability to generate **examples** and to explain every step. In this regard, the AI tool is much like a human tutor!

Example

In our demo chat, we see that the AI was able to explain the `to_datetime()` function to us in great detail, including several examples of when you might need to use common optional arguments like `format` or `errors`.

5.4.0.3 3. As a **search tool**, when you want to find the perfect function.

The beauty of packages like `pandas` is that many, many functions are provided to give you "shortcuts" for common data tasks. The curse is that you don't always know if a particular function exists!

Certainly, it is possible to look in official documentation - but this can be tedious and involve guessing which functions *might* solve your problem by their name alone. A better option might be a static search service, like Google - but to trigger useful results, you often need to use key words that precisely describe what you are looking for.

Since genAI tools can interpret human-style text, they are often able to understand what you are looking for and help you find the perfect function.

Example

In response to our question about misspellings in the dataset, the genAI offered two solutions: a simple one for when the possible misspellings are known, and a complex one for general use. In the second solution, it suggested an additional library called `fuzzywuzzy`.

5.4.0.4 4. As a **debugger**, when you can spot where in your analysis something is going wrong.

Example

Here we wanted to take the mean of a column called `num_admitted`, grouped by `hour` and `severity`. Unfortunately we made an extremely common mistake of putting the column name inside the `mean()` function. The genAI tool was able to explain why this approach is wrong, and offer possible solutions.

5.4.1 Cautionary tales and advice

5.4.1.1 Debugging works best on small pieces

Suppose you write out your full end-to-end data analysis, without testing or checking any intermediate steps, and it does not work as you hoped. Would you put the whole thing into an LLM for help?

While this may work in some straightforward cases, it also may confuse and frustrate you more, as the AI suggests a long list of possible issues for you to comb through.

Instead, we recommend stepping through your analysis line by line, until you reach a step where the results are not what you intended. (This is good troubleshooting strategy already, and you might find your bug before you even turn to an AI tool!) Then, you can ask the AI about only one or two lines of code that aren't working, and get a more direct and helpful answer.

5.4.1.2 Beware of hallucinations

Keep in mind that a genAI response is not the same as a systematic search: it is producing output trying to mimic what it has seen in the training data.

This means that if you ask for a function for a particular task, an AI tool **might** simply *invent* a function that seems similar to others it has seen in training. As AI tools are improving, and training data sources for python code are increasing, hallucinations about code are becoming quite rare. However, it's important to be aware that AI-produced suggestions are not equivalent to finding the function in official documentation; you'll need to verify for yourself.

Edit this page