Open in Colab

View source code | Read notebook in online book format

A Quick Introduction to Data Analysis and Manipulation with Python and pandas

```
In [1]: import datetime
        print(f"Last updated: {datetime.datetime.now()}")
```

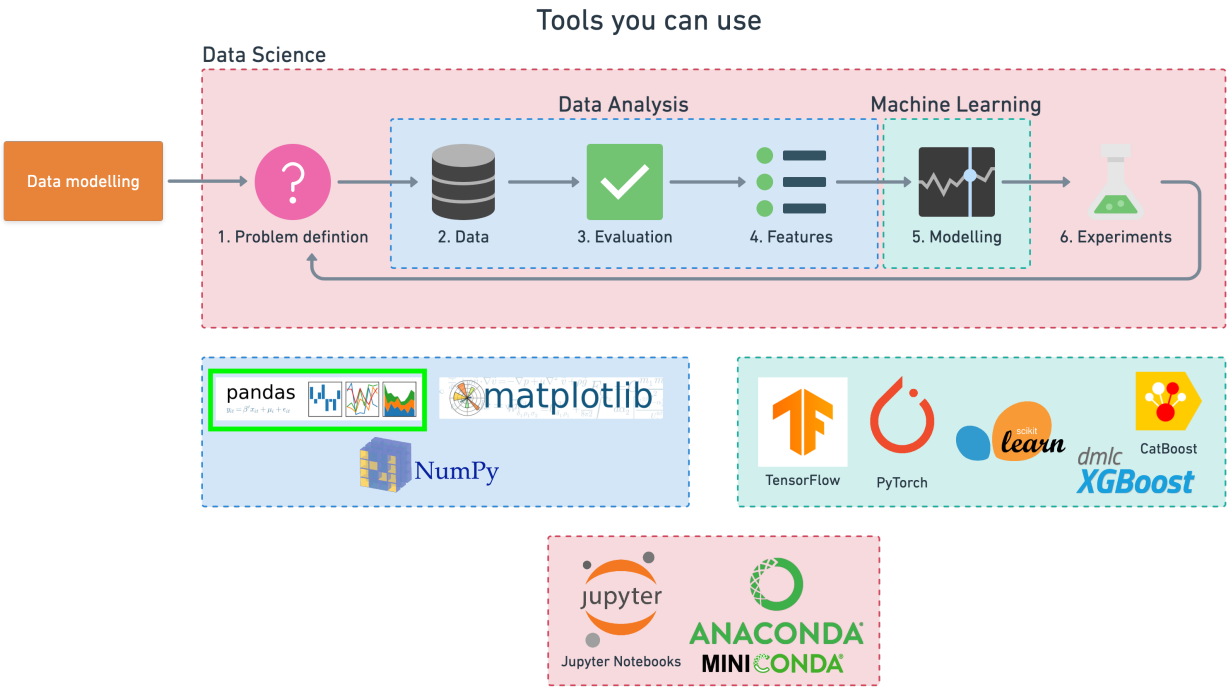
```
Last updated: 2024-05-17 17:19:29.997540
```

What is pandas?

If you're getting into machine learning and data science and you're using Python, you're going to use pandas.

pandas is an open source library which helps you analyse and manipulate data.

Table of contents



Why pandas?

pandas provides a simple to use but very capable set of functions you can use to on your data.

It's integrated with many other data science and machine learning tools which use Python so having an understanding of it will be helpful throughout your journey.

One of the main use cases you'll come across is using pandas to transform your data in a way which makes it usable with machine learning algorithms.

What does this notebook cover?

Because the pandas library is vast, there's often many ways to do the same thing. This notebook covers some of the most fundamental functions of the library, which are more than enough to get started.

Where can I get help?

If you get stuck or think of something you'd like to do which this notebook doesn't cover, don't fear!

The recommended steps you take are:

1. **Try it** - Since pandas is very friendly, your first step should be to use what you know and try figure out the answer to your own question (getting it wrong is part of the process). If in doubt, run your code.
2. **Search for it** - If trying it on your own doesn't work, since someone else has probably tried to do something similar, try searching for your problem in the following places (either via a search engine or direct):
    - pandas documentation - the best place for learning pandas, this resource covers all of the pandas functionality.

- Stack Overflow - this is the developers Q&A hub, it's full of questions and answers of different problems across a wide range of software development topics and chances are, there's one related to your problem.
- ChatGPT - ChatGPT is very good at explaining code, however, it can make mistakes. Best to verify the code it writes first before using it. Try asking "Can you explain the following code for me? {your code here}" and then continue with follow up questions from there.

An example of searching for a pandas function might be:

> "how to fill all the missing values of two columns using pandas"

Searching this on Google leads to this post on Stack Overflow:
https://stackoverflow.com/questions/36556256/how-do-i-fill-na-values-in-multiple-columns-in-pandas

The next steps here are to read through the post and see if it relates to your problem. If it does, great, take the code/information you need and **rewrite it** to suit your own problem.

3. **Ask for help** - If you've been through the above 2 steps and you're still stuck, you might want to ask your question on Stack Overflow. Remember to be specific as possible and provide details on what you've tried.

Remember, you don't have to learn all of these functions off by heart to begin with.

What's most important is remembering to continually ask yourself, "what am I trying to do with the data?".

Start by answering that question and then practicing finding the code which does it.

Let's get started.

0. Importing pandas

To get started using pandas, the first step is to import it.

The most common way (and method you should use) is to import pandas as the abbreviation `pd` (e.g. `pandas` -> `pd`).

If you see the letters `pd` used anywhere in machine learning or data science, it's probably referring to the pandas library.

In [2]:
```
import pandas as pd

# Print the version
print(f"pandas version: {pd.__version__}")
```

```
pandas version: 2.2.2
```

1. Datatypes

pandas has two main datatypes, `Series` and `DataFrame`.

- `pandas.Series` - a 1-dimensional column of data.
- `pandas.DataFrame` (most common) - a 2-dimesional table of data with rows and columns.

You can create a `Series` using `pd.Series()` and passing it a Python list.

In [3]:
```
# Creating a series of car types
cars = pd.Series(["BMW", "Toyota", "Honda"])
cars
```

Out[3]:
```
0       BMW
1    Toyota
2     Honda
dtype: object
```

In [4]:
```
# Creating a series of colours
colours = pd.Series(["Blue", "Red", "White"])
colours
```

Out[4]:
```
0     Blue
1      Red
2    White
dtype: object
```

You can create a `DataFrame` by using `pd.DataFrame()` and passing it a Python dictionary.

Let's use our two `Series` as the values.

In [5]:
```
# Creating a DataFrame of cars and colours
car_data = pd.DataFrame({"Car type": cars,
                         "Colour": colours})
car_data
```

Out[5]:

	Car type	Colour
0	BMW	Blue
1	Toyota	Red
2	Honda	White

You can see the keys of the dictionary became the column headings (text in bold) and the values of the two `Series`'s became the values in the DataFrame.

It's important to note, many different types of data could go into the DataFrame.

Here we've used only text but you could use floats, integers, dates and more.

Exercises

1. Make a `Series` of different foods.
2. Make a `Series` of different dollar values (these can be integers).
3. Combine your `Series`'s of foods and dollar values into a `DataFrame`.

Try it out for yourself first, then see how your code goes against the solution.

Note: Make sure your two `Series` are the same size before combining them in a DataFrame.

In [6]:
```
# Your code here
```

In [7]:
```
# Example solution

# Make a Series of different foods
foods = pd.Series(["Almond butter", "Eggs", "Avocado"])

# Make a Series of different dollar values
prices = pd.Series([9, 6, 2])

# Combine your Series of foods and dollar values into a DataFrame
food_data = pd.DataFrame({"Foods": foods,
```

```
                            "Price": prices})

food_data
```

Out[7]:

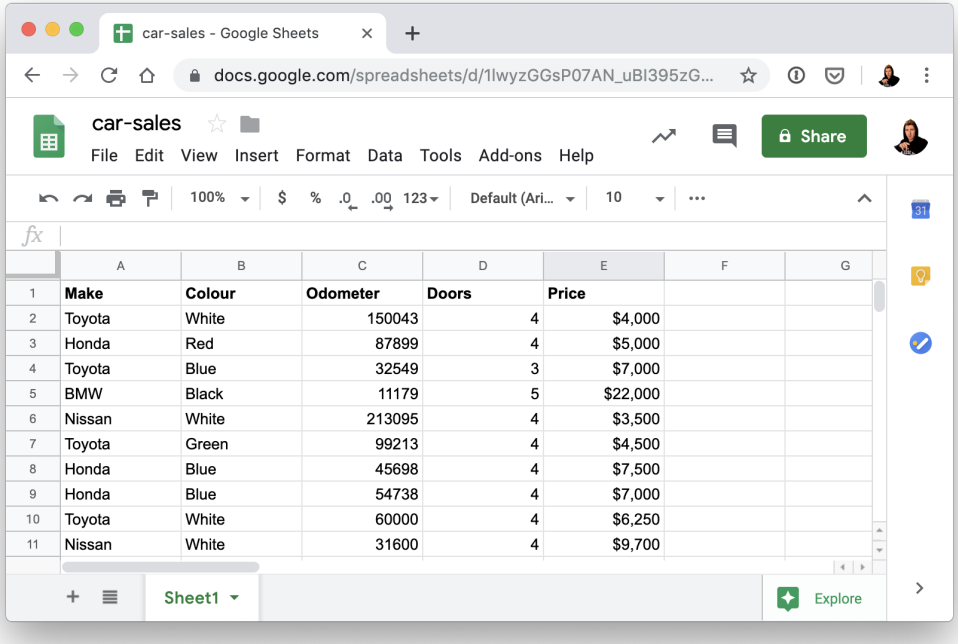
	Foods	Price
0	Almond butter	9
1	Eggs	6
2	Avocado	2

2. Importing data

Creating `Series` and `DataFrame`'s from scratch is nice but what you'll usually be doing is importing your data in the form of a `.csv` (comma separated value), spreadsheet file or something similar such as an SQL database.

pandas allows for easy importing of data like this through functions such as `pd.read_csv()` and `pd.read_excel()` (for Microsoft Excel files).

Say you wanted to get this information from this Google Sheet document into a pandas `DataFrame`.



You could export it as a `.csv` file and then import it using `pd.read_csv()`.

> **Tip:** If the Google Sheet is public, `pd.read_csv()` can read it via URL, try searching for "pandas read Google Sheet with URL".

In this case, the exported `.csv` file is called `car-sales.csv`.

In [8]:
```
# Import car sales data
car_sales = pd.read_csv("../data/car-sales.csv") # takes a filename as string as i

# Option 2: Read directly from a URL/Google Sheets
```

```
# If you are reading from GitHub, be sure to use the "raw" link (original link: htt
car_sales = pd.read_csv("https://raw.githubusercontent.com/mrdbourke/zero-to-mast
car_sales
```

Out[8]:

	Make	Colour	Odometer (KM)	Doors	Price
0	Toyota	White	150043	4	$4,000.00
1	Honda	Red	87899	4	$5,000.00
2	Toyota	Blue	32549	3	$7,000.00
3	BMW	Black	11179	5	$22,000.00
4	Nissan	White	213095	4	$3,500.00
5	Toyota	Green	99213	4	$4,500.00
6	Honda	Blue	45698	4	$7,500.00
7	Honda	Blue	54738	4	$7,000.00
8	Toyota	White	60000	4	$6,250.00
9	Nissan	White	31600	4	$9,700.00

Now we've got the same data from the spreadsheet available in a pandas `DataFrame` called `car_sales`.

Having your data available in a `DataFrame` allows you to take advantage of all of pandas functionality on it.

Another common practice you'll see is data being imported to `DataFrame` called `df` (short for `DataFrame`).

In [9]:

```
# Import the car sales data and save it to df

# Option 1: Read from a CSV file (stored on our local computer)
df = pd.read_csv("../data/car-sales.csv")
```

```
# Option 2: Read directly from a URL/Google Sheets (if the file is hosted online)
df = pd.read_csv("https://raw.githubusercontent.com/mrdbourke/zero-to-mastery-ml/
df
```

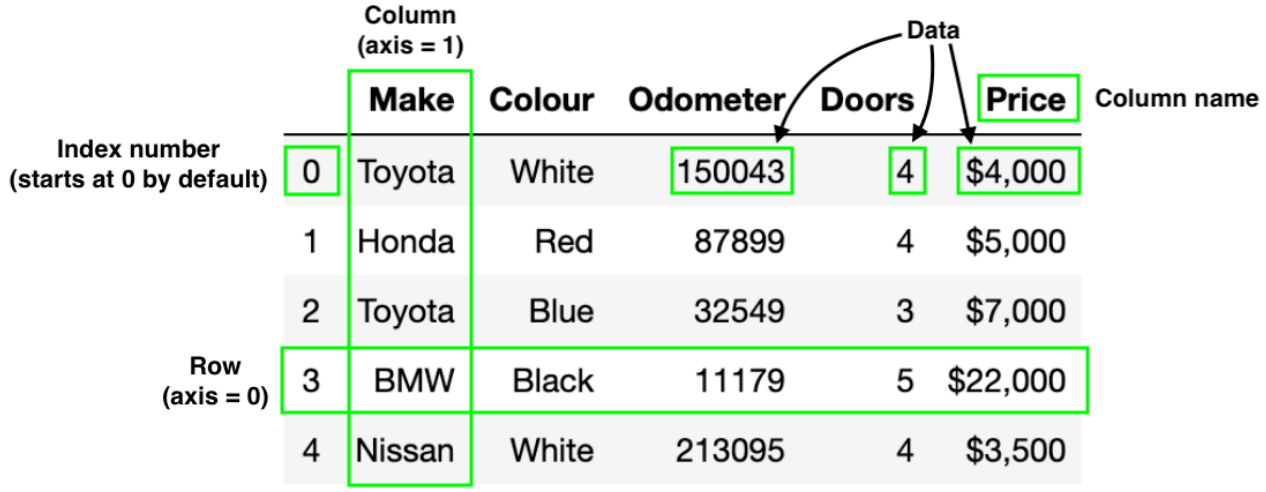
Out[9]:

	Make	Colour	Odometer (KM)	Doors	Price
0	Toyota	White	150043	4	$4,000.00
1	Honda	Red	87899	4	$5,000.00
2	Toyota	Blue	32549	3	$7,000.00
3	BMW	Black	11179	5	$22,000.00
4	Nissan	White	213095	4	$3,500.00
5	Toyota	Green	99213	4	$4,500.00
6	Honda	Blue	45698	4	$7,500.00
7	Honda	Blue	54738	4	$7,000.00
8	Toyota	White	60000	4	$6,250.00
9	Nissan	White	31600	4	$9,700.00

Now `car_sales` and `df` contain the exact same information, the only difference is the name. Like any other variable, you can name your `DataFrame`'s whatever you want. But best to choose something simple.

Anatomy of a DataFrame

Different functions use different labels for different things. This graphic sums up some of the main components of `DataFrame`'s and their different names.



3. Exporting data

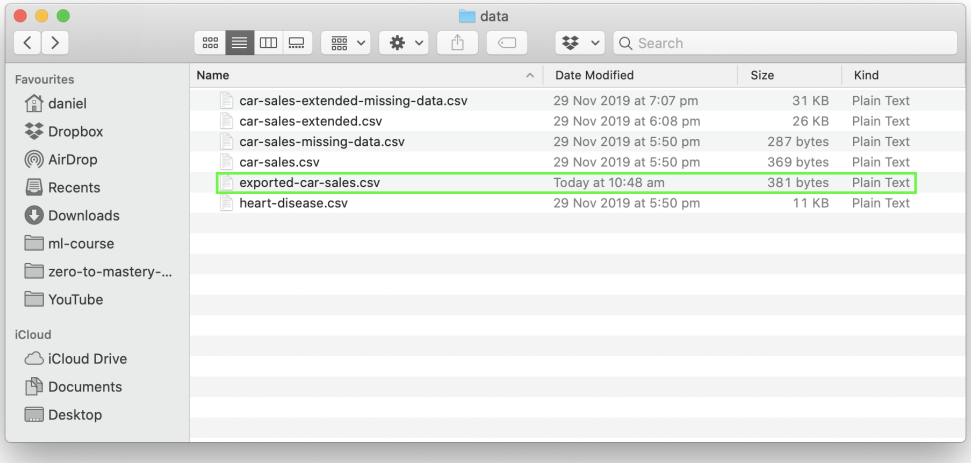
After you've made a few changes to your data, you might want to export it and save it so someone else can access the changes.

pandas allows you to export `DataFrame`'s to `.csv` format using `.to_csv()` or spreadsheet format using `.to_excel()`.

We haven't made any changes yet to the `car_sales` `DataFrame` but let's try export it.

```
In [10]: # Export the car sales DataFrame to csv
         car_sales.to_csv("../data/exported-car-sales.csv")
```

Running this will save a file called `export-car-sales.csv` to the current folder.



Exercises

1. Practice importing a `.csv` file using `pd.read_csv()`, you can download `heart-disease.csv`. This file contains annonymous patient medical records and whether or not they have heart disease.
2. Practice exporting a `DataFrame` using `.to_csv()`. You could export the heart disease `DataFrame` after you've imported it.

Note:

- Make sure the `heart-disease.csv` file is in the same folder as your notebook orbe sure to use the filepath where the file is.
- You can name the variables and exported files whatever you like but make sure they're readable.

In [11]:
```
# Your code here
```

Example solution

In [12]:
```
# Importing heart-disease.csv
patient_data = pd.read_csv("../data/heart-disease.csv")
patient_data
```

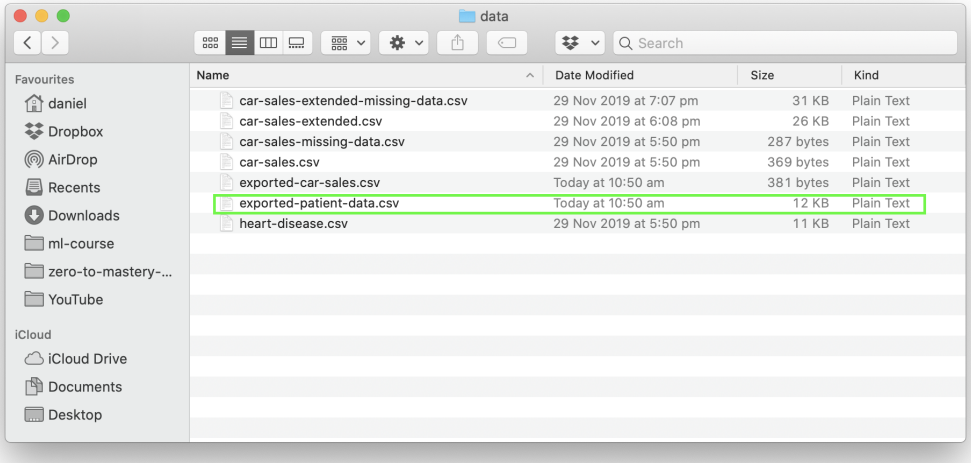
Out[12]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1	1
1	37	1	2	130	250	0	1	187	0	3.5	0	0	2	1
2	41	0	1	130	204	0	0	172	0	1.4	2	0	2	1
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2	1
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
298	57	0	0	140	241	0	1	123	1	0.2	1	0	3	0
299	45	1	3	110	264	0	1	132	0	1.2	1	0	3	0
300	68	1	0	144	193	1	1	141	0	3.4	1	2	3	0

301	57	1	0	130	131	0	1	115	1	1.2	1	1	3	0
302	57	0	1	130	236	0	0	174	0	0.0	1	1	2	0

303 rows × 14 columns

In [13]:
```
# Exporting the patient_data DataFrame to csv
patient_data.to_csv("../data/exported-patient-data.csv")
```



4. Describing data

One of the first things you'll want to do after you import some data into a pandas `DataFrame` is to start exploring it.

pandas has many built in functions which allow you to quickly get information about a `DataFrame`.

Let's explore some using the `car_sales` `DataFrame`.

In [14]:
```
car_sales
```

Out[14]:

	Make	Colour	Odometer (KM)	Doors	Price
0	Toyota	White	150043	4	$4,000.00
1	Honda	Red	87899	4	$5,000.00
2	Toyota	Blue	32549	3	$7,000.00
3	BMW	Black	11179	5	$22,000.00
4	Nissan	White	213095	4	$3,500.00
5	Toyota	Green	99213	4	$4,500.00
6	Honda	Blue	45698	4	$7,500.00
7	Honda	Blue	54738	4	$7,000.00
8	Toyota	White	60000	4	$6,250.00
9	Nissan	White	31600	4	$9,700.00

`.dtypes` shows us what datatype each column contains.

In [15]:
```
car_sales.dtypes
```

Out[15]:
```
Make             object
Colour           object
Odometer (KM)     int64
Doors             int64
Price            object
dtype: object
```

Notice how the `Price` column isn't an integer like `Odometer` or `Doors`. Don't worry, pandas makes this easy to fix.

`.describe()` gives you a quick statistical overview of the numerical columns.

In [16]:
```
car_sales.describe()
```

Out[16]:

	Odometer (KM)	Doors
count	10.000000	10.000000
mean	78601.400000	4.000000
std	61983.471735	0.471405
min	11179.000000	3.000000
25%	35836.250000	4.000000
50%	57369.000000	4.000000
75%	96384.500000	4.000000
max	213095.000000	5.000000

`.info()` shows a handful of useful information about a `DataFrame` such as:

- How many entries (rows) there are
- Whether there are missing values (if a columns non-null value is less than the number of entries, it has missing values)
- The datatypes of each column

In [17]:
```
car_sales.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 5 columns):
 #   Column          Non-Null Count  Dtype 
---  ------          --------------  ----- 
 0   Make            10 non-null     object
 1   Colour          10 non-null     object
 2   Odometer (KM)   10 non-null     int64 
 3   Doors           10 non-null     int64 
 4   Price           10 non-null     object
dtypes: int64(2), object(3)
memory usage: 532.0+ bytes
```

You can also call various statistical and mathematical methods such as `.mean()` or `.sum()` directly on a `DataFrame` or `Series`.

In [18]:
```
# Calling .mean() on a DataFrame
car_sales.mean(numeric_only=True) # numeric_only = get mean values of numeric colu
```

Out[18]:
```
Odometer (KM)    78601.4
Doors                4.0
dtype: float64
```

In [19]:
```
# Calling .mean() on a Series
car_prices = pd.Series([3000, 3500, 11250])
car_prices.mean()
```

Out[19]:
```
5916.666666666667
```

In [20]:
```
# Calling .sum() on a DataFrame with numeric_only=False (default)
car_sales.sum(numeric_only=False)
```

Out[20]:
```
Make             ToyotaHondaToyotaBMWNissanToyotaHondaHondaToyo...
Colour               WhiteRedBlueBlackWhiteGreenBlueBlueWhiteWhite
```

```
Odometer (KM)                                                786014
Doors                                                            40
Price             $4,000.00$5,000.00$7,000.00$22,000.00$3,500.00...
dtype: object
```

In [21]:
```python
# Calling .sum() on a DataFrame with numeric_only=True
car_sales.sum(numeric_only=True)
```

Out[21]:
```
Odometer (KM)    786014
Doors                40
dtype: int64
```

In [22]:
```python
# Calling .sum() on a Series
car_prices.sum()
```

Out[22]:
```
17750
```

Calling these on a whole `DataFrame` may not be as helpful as targeting an individual column. But it's helpful to know they're there.

`.columns` will show you all the columns of a `DataFrame`.

In [23]:
```python
car_sales.columns
```

Out[23]:
```
Index(['Make', 'Colour', 'Odometer (KM)', 'Doors', 'Price'], dtype='object')
```

You can save them to a list which you could use later.

In [24]:
```python
# Save car_sales columns to a list
car_columns = car_sales.columns
car_columns[0]
```

```
Out[24]: 'Make'
```

`.index` will show you the values in a `DataFrame`'s index (the column on the far left).

```
In [25]: car_sales.index
```

```
Out[25]: RangeIndex(start=0, stop=10, step=1)
```

pandas `DataFrame`'s, like Python lists, are 0-indexed (unless otherwise changed). This means they start at 0.

	Make	Colour	Odometer	Doors	Price
Index number (starts at 0 by default) → 0	Toyota	White	150043	4	$4,000
1	Honda	Red	87899	4	$5,000
2	Toyota	Blue	32549	3	$7,000
3	BMW	Black	11179	5	$22,000
4	Nissan	White	213095	4	$3,500

```
In [26]: # Show the length of a DataFrame
         len(car_sales)
```

```
Out[26]: 10
```

So even though the length of our `car_sales` dataframe is 10, this means the indexes go from 0-9.

5. Viewing and selecting data

Some common methods for viewing and selecting data in a pandas DataFrame include:

- `DataFrame.head(n=5)` - Displays the first `n` rows of a DataFrame (e.g. `car_sales.head()` will show the first 5 rows of the `car_sales` DataFrame).
- `DataFrame.tail(n=5)` - Displays the last `n` rows of a DataFrame.
- `DataFrame.loc[]` - Accesses a group of rows and columns by labels or a boolean array.
- `DataFrame.iloc[]` - Accesses a group of rows and columns by integer indices (e.g. `car_sales.iloc[0]` shows all the columns from index `0`.
- `DataFrame.columns` - Lists the column labels of the DataFrame.
- `DataFrame['A']` - Selects the column named `'A'` from the DataFrame.
- `DataFrame[DataFrame['A'] > 5]` - Boolean indexing filters rows based on column values meeting a condition (e.g. all rows from column `'A'` greater than `5`.
- `DataFrame.plot()` - Creates a line plot of a DataFrame's columns (e.g. plot `Make` vs. `Odometer (KM)` columns with `car_sales["Make", "Odometer (KM)"]].plot();`).
- `DataFrame.hist()` - Generates histograms for columns in a DataFrame.
- `pandas.crosstab()` - Computes a cross-tabulation of two or more factors.

In practice, you'll constantly be making changes to your data, and viewing it. Changing it, viewing it, changing it, viewing it.

You won't always want to change all of the data in your `DataFrame`'s either. So there are just as many different ways to select data as there is to view it.

`.head()` allows you to view the first 5 rows of your `DataFrame`. You'll likely be using this one a lot.

In [27]:
```
# Show the first 5 rows of car_sales
car_sales.head()
```

Out[27]:

	Make	Colour	Odometer (KM)	Doors	Price
0	Toyota	White	150043	4	$4,000.00
1	Honda	Red	87899	4	$5,000.00
2	Toyota	Blue	32549	3	$7,000.00
3	BMW	Black	11179	5	$22,000.00
4	Nissan	White	213095	4	$3,500.00

Why 5 rows? Good question. I don't know the answer. But 5 seems like a good amount.

Want more than 5?

No worries, you can pass `.head()` an integer to display more than or less than 5 rows.

In [28]:
```
# Show the first 7 rows of car_sales
car_sales.head(7)
```

Out[28]:

	Make	Colour	Odometer (KM)	Doors	Price
0	Toyota	White	150043	4	$4,000.00

1	Honda	Red	87899	4	$5,000.00
2	Toyota	Blue	32549	3	$7,000.00
3	BMW	Black	11179	5	$22,000.00
4	Nissan	White	213095	4	$3,500.00
5	Toyota	Green	99213	4	$4,500.00
6	Honda	Blue	45698	4	$7,500.00

`.tail()` allows you to see the bottom 5 rows of your `DataFrame`. This is helpful if your changes are influencing the bottom rows of your data.

In [29]:
```
# Show bottom 5 rows of car_sales
car_sales.tail()
```

Out[29]:

	Make	Colour	Odometer (KM)	Doors	Price
5	Toyota	Green	99213	4	$4,500.00
6	Honda	Blue	45698	4	$7,500.00
7	Honda	Blue	54738	4	$7,000.00
8	Toyota	White	60000	4	$6,250.00
9	Nissan	White	31600	4	$9,700.00

You can use `.loc[]` and `.iloc[]` to select data from your `Series` and `DataFrame`'s.

Let's see.

In [30]:
```
# Create a sample series
animals = pd.Series(["cat", "dog", "bird", "snake", "ox", "lion"],
                    index=[0, 3, 9, 8, 67, 3])
animals
```

Out[30]:
```
0       cat
3       dog
9      bird
8     snake
67       ox
3      lion
dtype: object
```

`.loc[]` takes an integer or label as input. And it chooses from your `Series` or `DataFrame` whichever index matches the number.

In [31]:
```
# Select all indexes with 3
animals.loc[3]
```

Out[31]:
```
3     dog
3    lion
dtype: object
```

In [32]:
```
# Select index 9
animals.loc[9]
```

Out[32]:
```
'bird'
```

Let's try with our `car_sales` DataFrame.

In [33]:
```
car_sales
```

Out[33]:

	Make	Colour	Odometer (KM)	Doors	Price
0	Toyota	White	150043	4	$4,000.00
1	Honda	Red	87899	4	$5,000.00
2	Toyota	Blue	32549	3	$7,000.00

3	BMW	Black	11179	5	$22,000.00
4	Nissan	White	213095	4	$3,500.00
5	Toyota	Green	99213	4	$4,500.00
6	Honda	Blue	45698	4	$7,500.00
7	Honda	Blue	54738	4	$7,000.00
8	Toyota	White	60000	4	$6,250.00
9	Nissan	White	31600	4	$9,700.00

In [34]:
```
# Select row at index 3
car_sales.loc[3]
```

Out[34]:
```
Make                  BMW
Colour              Black
Odometer (KM)       11179
Doors                   5
Price            $22,000.00
Name: 3, dtype: object
```

`iloc[]` does a similar thing but works with exact positions.

In [35]:
```
animals
```

Out[35]:
```
0       cat
3       dog
9      bird
8     snake
67       ox
3      lion
dtype: object
```

In [36]:
```
# Select row at position 3
animals.iloc[3]
```

```
Out[36]: 'snake'
```

Even though `'snake'` appears at index 8 in the series, it's shown using `.iloc[3]` because it's at the 3rd (starting from 0) position.

Let's try with the `car_sales` `DataFrame`.

```
In [37]: # Select row at position 3
         car_sales.iloc[3]
```

```
Out[37]: Make                  BMW
         Colour              Black
         Odometer (KM)       11179
         Doors                   5
         Price            $22,000.00
         Name: 3, dtype: object
```

You can see it's the same as `.loc[]` because the index is in order, position 3 is the same as index 3.

You can also use slicing with `.loc[]` and `.iloc[]`.

```
In [38]: # Get all rows up to position 3
         animals.iloc[:3]
```

```
Out[38]: 0     cat
         3     dog
         9    bird
         dtype: object
```

```
In [39]: # Get all rows up to (and including) index 3
         car_sales.loc[:3]
```

```
Out[39]:
```

	Make	Colour	Odometer (KM)	Doors	Price

0	Toyota	White	150043	4	$4,000.00
1	Honda	Red	87899	4	$5,000.00
2	Toyota	Blue	32549	3	$7,000.00
3	BMW	Black	11179	5	$22,000.00

In [40]:
```
# Get all rows of the "Colour" column
car_sales.loc[:, "Colour"] # note: ":" stands for "all", e.g. "all indices in the
```

Out[40]:
```
0    White
1      Red
2     Blue
3    Black
4    White
5    Green
6     Blue
7     Blue
8    White
9    White
Name: Colour, dtype: object
```

When should you use `.loc[]` or `.iloc[]`?

- Use `.loc[]` when you're selecting rows and columns **based on their lables or a condition** (e.g. retrieving data for specific columns).
- Use `.iloc[]` when you're selecting rows and columns **based on their integer index positions** (e.g. extracting the first ten rows regardless of the labels).

However, in saying this, it will often take a bit of practice with each of the methods before you figure out which you'd like to use.

If you want to select a particular column, you can use `DataFrame['COLUMN_NAME']`.

In [41]:
```
# Select Make column
car_sales['Make']
```

Out[41]:
```
0    Toyota
1     Honda
2    Toyota
3       BMW
4    Nissan
5    Toyota
6     Honda
7     Honda
8    Toyota
9    Nissan
Name: Make, dtype: object
```

In [42]:
```
# Select Colour column
car_sales['Colour']
```

Out[42]:
```
0    White
1      Red
2     Blue
3    Black
4    White
5    Green
6     Blue
7     Blue
8    White
9    White
Name: Colour, dtype: object
```

Boolean indexing works with column selection too. Using it will select the rows which fulfill the condition in the brackets.

In [43]:
```
# Select cars with over 100,000 on the Odometer
car_sales[car_sales["Odometer (KM)"] > 100000]
```

Out[43]:

	Make	Colour	Odometer (KM)	Doors	Price
0	Toyota	White	150043	4	$4,000.00
4	Nissan	White	213095	4	$3,500.00

In [44]:
```
# Select cars which are made by Toyota
car_sales[car_sales["Make"] == "Toyota"]
```

Out[44]:

	Make	Colour	Odometer (KM)	Doors	Price
0	Toyota	White	150043	4	$4,000.00
2	Toyota	Blue	32549	3	$7,000.00
5	Toyota	Green	99213	4	$4,500.00
8	Toyota	White	60000	4	$6,250.00

`pd.crosstab()` is a great way to view two different columns together and compare them.

In [45]:
```
# Compare car Make with number of Doors
pd.crosstab(car_sales["Make"], car_sales["Doors"])
```

Out[45]:

Doors	3	4	5
Make			
BMW	0	0	1

Honda	0	3	0
Nissan	0	2	0
Toyota	1	3	0

If you want to compare more columns in the context of another column, you can use `.groupby()`.

In [46]:
```
car_sales
```

Out[46]:

	Make	Colour	Odometer (KM)	Doors	Price
0	Toyota	White	150043	4	$4,000.00
1	Honda	Red	87899	4	$5,000.00
2	Toyota	Blue	32549	3	$7,000.00
3	BMW	Black	11179	5	$22,000.00
4	Nissan	White	213095	4	$3,500.00
5	Toyota	Green	99213	4	$4,500.00
6	Honda	Blue	45698	4	$7,500.00
7	Honda	Blue	54738	4	$7,000.00
8	Toyota	White	60000	4	$6,250.00
9	Nissan	White	31600	4	$9,700.00

In [47]:
```
# Group by the Make column and find the mean of the other columns
car_sales.groupby(["Make"]).mean(numeric_only=True)
```

Out[47]:

	Odometer (KM)	Doors
Make		
BMW	11179.000000	5.00
Honda	62778.333333	4.00

Nissan	122347.500000	4.00
Toyota	85451.250000	3.75

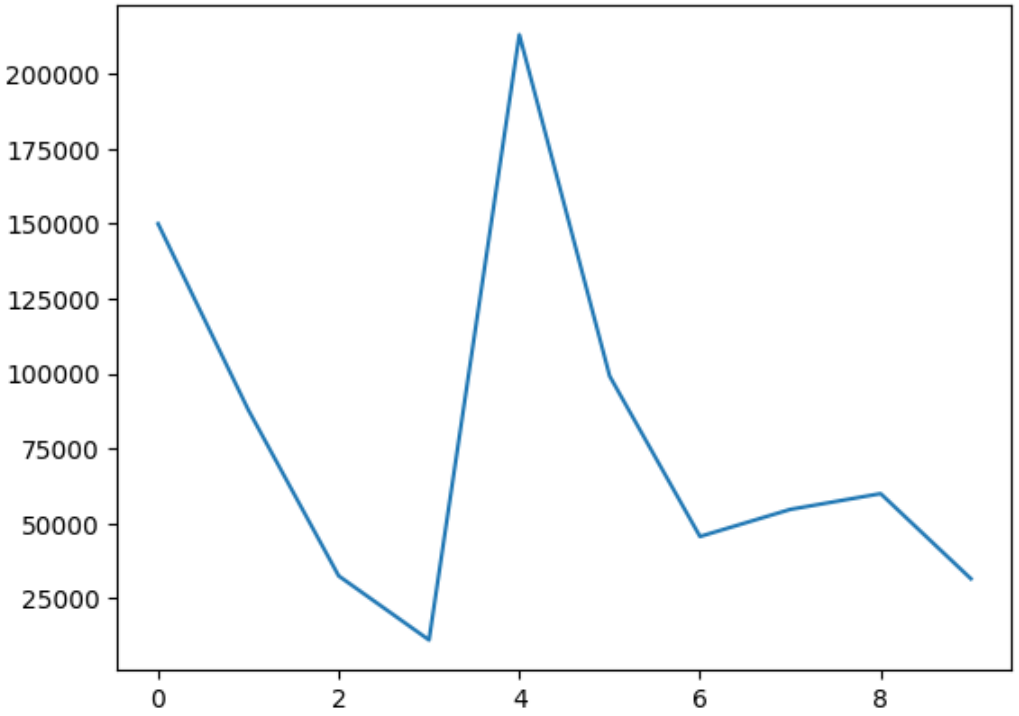
pandas even allows for quick plotting of columns so you can see your data visualling. To plot, you'll have to import `matplotlib`. If your plots aren't showing, try running the two lines of code below.

`%matplotlib inline` is a special command which tells Jupyter to show your plots. Commands with `%` at the front are called magic commands.

In [48]:
```
# Import matplotlib and tell Jupyter to show plots
import matplotlib.pyplot as plt
%matplotlib inline
```

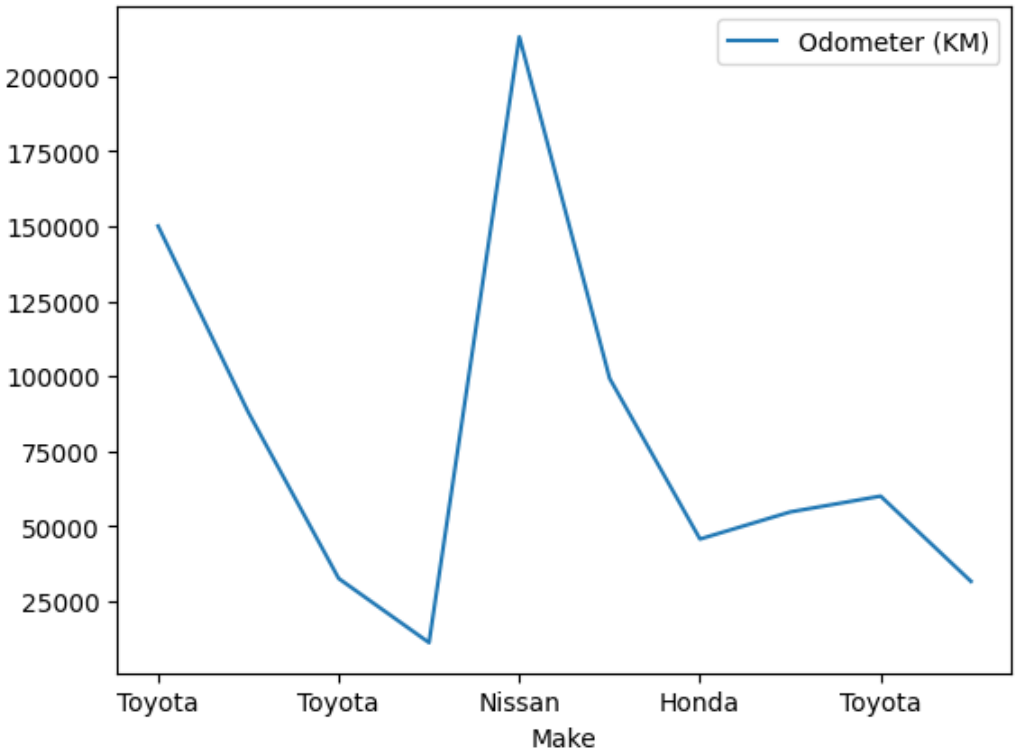
You can visualize a column by calling `.plot()` on it.

In [49]:
```
car_sales["Odometer (KM)"].plot(); # tip: the ";" on the end prevents matplotlib f
```



Or compare two columns by passing them as `x` and `y` to `plot()`.

```
In [50]: car_sales.plot(x="Make", y="Odometer (KM)");
```

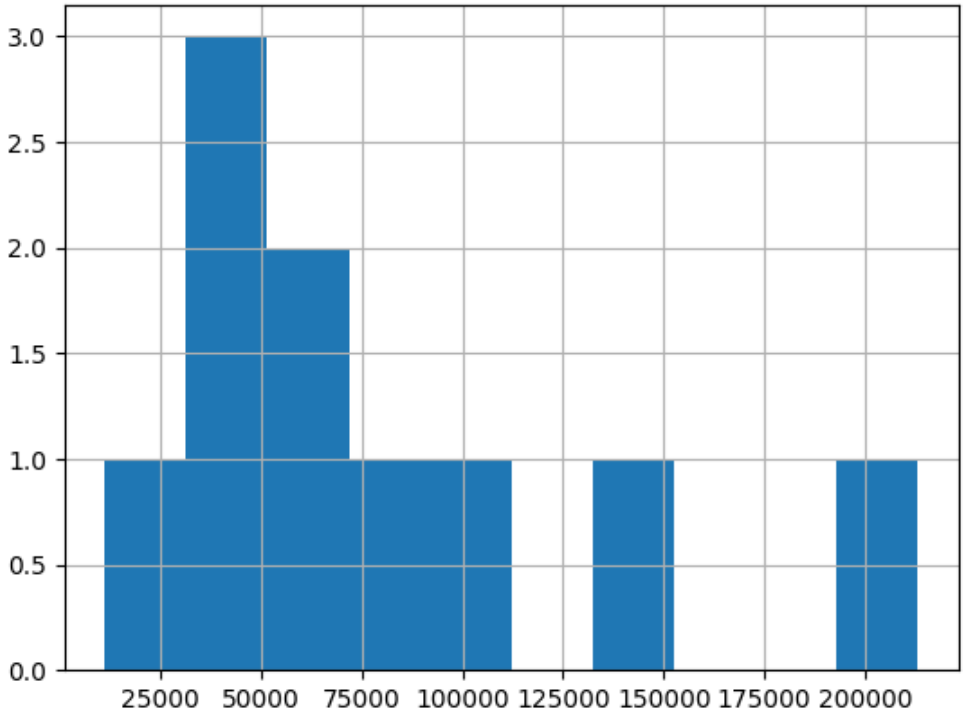


You can see the distribution of a column by calling `.hist()` on you.

The distribution of something is a way of describing the spread of different values.

```
In [51]: car_sales["Odometer (KM)"].hist()
```

```
Out[51]: <Axes: >
```



In this case, the majority of the **distribution** (spread) of the `"Odometer (KM)"` column is more towards the left of the graph. And there are two values which are more to the right. These two values to the right could be considered **outliers** (not part of the majority).

Now what if we wanted to plot our `"Price"` column?

Let's try.

In [52]:
```
car_sales["Price"].plot()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[52], line 1
----> 1 car_sales["Price"].plot()

File ~/miniforge3/envs/ztm-ml-env/lib/python3.11/site-packages/pandas/plotting/_cor
e.py:1030, in PlotAccessor.__call__(self, *args, **kwargs)
   1027             label_name = label_kw or data.columns
   1028             data.columns = label_name
-> 1030 return plot_backend.plot(data, kind=kind, **kwargs)

File ~/miniforge3/envs/ztm-ml-env/lib/python3.11/site-packages/pandas/plotting/_mat
plotlib/__init__.py:71, in plot(data, kind, **kwargs)
     69         kwargs["ax"] = getattr(ax, "left_ax", ax)
     70 plot_obj = PLOT_CLASSES[kind](data, **kwargs)
---> 71 plot_obj.generate()
     72 plot_obj.draw()
     73 return plot_obj.result

File ~/miniforge3/envs/ztm-ml-env/lib/python3.11/site-packages/pandas/plotting/_mat
plotlib/core.py:499, in MPLPlot.generate(self)
    497 @final
    498 def generate(self) -> None:
--> 499     self._compute_plot_data()
    500     fig = self.fig
    501     self._make_plot(fig)

File ~/miniforge3/envs/ztm-ml-env/lib/python3.11/site-packages/pandas/plotting/_mat
plotlib/core.py:698, in MPLPlot._compute_plot_data(self)
    696 # no non-numeric frames or series allowed
    697 if is_empty:
--> 698     raise TypeError("no numeric data to plot")
    700 self.data = numeric_data.apply(type(self)._convert_to_ndarray)

TypeError: no numeric data to plot
```

Trying to run it leaves us with an error. This is because the `"Price"` column of `car_sales` isn't in numeric form. We can tell this because of the `TypeError: no numeric data to plot` at the bottom of the cell.

We can check this with `.info()`.

```
In [53]: car_sales.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 5 columns):
 #   Column          Non-Null Count  Dtype 
---  ------          --------------  ----- 
 0   Make            10 non-null     object
 1   Colour          10 non-null     object
 2   Odometer (KM)   10 non-null     int64 
 3   Doors           10 non-null     int64 
 4   Price           10 non-null     object
dtypes: int64(2), object(3)
memory usage: 532.0+ bytes
```

So what can we do?

We need to convert the `"Price"` column to a numeric type.

How?

We could try a few different things on our own. But let's practice researching.

1. Open up a search engine and type in something like "how to convert a pandas column price to integer".

In the first result, I found this Stack Overflow question and answer . Where someone has had the same problem as us and someone else has provided an answer.

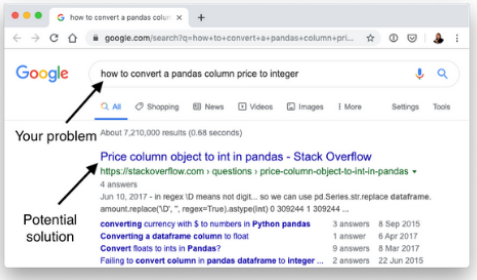
> **Note:** Sometimes the answer you're looking for won't be in the first result, or the 2nd or the 3rd. You may have to combine a few different solutions. Or, if possible, you can try and ask ChatGPT to help you out.

2. In practice, you'd read through this and see if it relates to your problem.

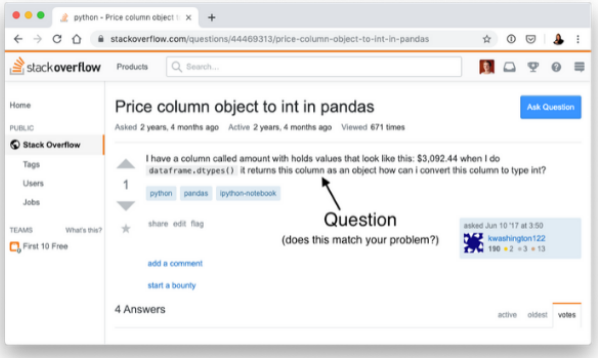
3. If it does, you can adjust the code from what's given in the Stack Overflow answer(s) to your own problem.

4. If you're still stuck, you can try and converse with ChatGPT to help you with your problem (as long as the data/problem you're working on is okay to share - never share private data with anyone on the internet, including AI chatbots).

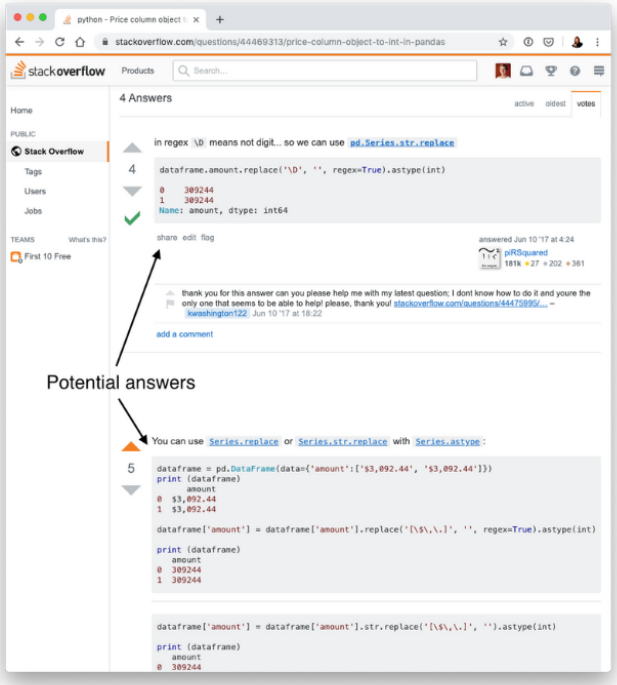
1



2



3



What's important in the beginning is not to remember every single detail off by heart but to know where to look. Remember, if in doubt, write code, run it, see what happens.

Let's copy the answer code here and see how it relates to our problem.

Answer code: `dataframe['amount'] = dataframe['amount'].str.replace('[\$\,\.]', '').astype(int)`

There's a lot going on here but what we can do is change the parts which aren't in our problem and keep the rest the same.

Our `DataFrame` is called `car_sales` not `dataframe`.

`car_sales['amount'] = car_sales['amount'].str.replace('[\$\,\.]', '').astype(int)`

And our `'amount'` column is called `"Price"`.

`car_sales["Price"] = car_sales["Price"].str.replace('[\$\,\.]', '').astype(int)`

That looks better. What the code on the right of `car_sales["Price"]` is saying is "remove the $ sign and comma and change the type of the cell to int".

Let's see what happens.

In [54]:
```
# Change Price column to integers
car_sales["Price"] = car_sales["Price"].str.replace('[\$\,\.]', '', regex=True)
car_sales
```

Out[54]:

	Make	Colour	Odometer (KM)	Doors	Price
0	Toyota	White	150043	4	400000
1	Honda	Red	87899	4	500000
2	Toyota	Blue	32549	3	700000
3	BMW	Black	11179	5	2200000
4	Nissan	White	213095	4	350000
5	Toyota	Green	99213	4	450000
6	Honda	Blue	45698	4	750000
7	Honda	Blue	54738	4	700000
8	Toyota	White	60000	4	625000
9	Nissan	White	31600	4	970000

Cool! but there are extra zeros in the `Price` column.

Let's remove it.

In [55]:
```
# Remove 2 extra zeros from the price column (2200000 -> 22000) by indexing all but
car_sales["Price"] = car_sales["Price"].str[:-2].astype(int)
car_sales
```

Out[55]:

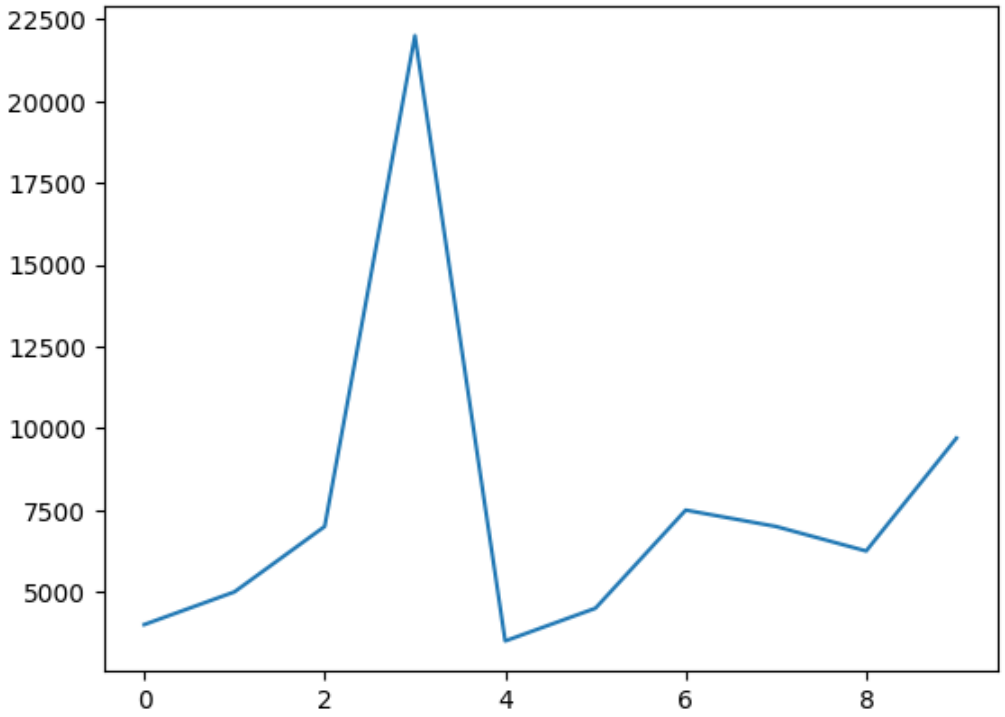
	Make	Colour	Odometer (KM)	Doors	Price
0	Toyota	White	150043	4	4000
1	Honda	Red	87899	4	5000
2	Toyota	Blue	32549	3	7000
3	BMW	Black	11179	5	22000
4	Nissan	White	213095	4	3500
5	Toyota	Green	99213	4	4500
6	Honda	Blue	45698	4	7500
7	Honda	Blue	54738	4	7000
8	Toyota	White	60000	4	6250
9	Nissan	White	31600	4	9700

In [56]:
```
car_sales.dtypes
```

Out[56]:
```
Make             object
Colour           object
Odometer (KM)     int64
Doors             int64
Price             int64
dtype: object
```

Beautiful! Now let's try to plot it agian.

In [57]:
```
car_sales["Price"].plot();
```



This is one of the many ways you can manipulate data using pandas.

When you see a number of different functions in a row, it's referred to as **chaining**. This means you add together a series of functions all to do one overall task.

Let's see a few more ways of manipulating data.

6. Manipulating data

You've seen an example of one way to manipulate data but pandas has many more.

How many more?

Put it this way, if you can imagine it, chances are, pandas can do it.

Let's start with string methods. Because pandas is based on Python, however you can manipulate strings in Python, you can do the same in pandas.

You can access the string value of a column using `.str`. Knowing this, how do you think you'd set a column to lowercase?

In [58]:
```
# Lower the Make column
car_sales["Make"].str.lower()
```

Out[58]:
```
0    toyota
1     honda
2    toyota
3       bmw
4    nissan
5    toyota
6     honda
7     honda
8    toyota
9    nissan
Name: Make, dtype: object
```

Notice how it doesn't change the values of the original `car_sales` `DataFrame` unless we set it equal to.

In [59]:
```
# View top 5 rows, Make column not lowered
car_sales.head()
```

Out[59]:

	Make	Colour	Odometer (KM)	Doors	Price
0	Toyota	White	150043	4	4000
1	Honda	Red	87899	4	5000
2	Toyota	Blue	32549	3	7000
3	BMW	Black	11179	5	22000
4	Nissan	White	213095	4	3500

In [60]:

```
# Set Make column to be lowered
car_sales["Make"] = car_sales["Make"].str.lower()
car_sales.head()
```

Out[60]:

	Make	Colour	Odometer (KM)	Doors	Price
0	toyota	White	150043	4	4000
1	honda	Red	87899	4	5000
2	toyota	Blue	32549	3	7000
3	bmw	Black	11179	5	22000
4	nissan	White	213095	4	3500

Reassigning the column changes it in the original `DataFrame`. This trend occurs throughout all kinds of data manipulation with pandas.

Some functions have a parameter called `inplace` which means a `DataFrame` is updated in place without having to reassign it.

Let's see what it looks like in combination with `.fillna()`, a function which fills missing data. But the thing is, our table isn't missing any data.

In practice, it's likely you'll work with datasets which aren't complete. What this means is you'll have to decide whether how to fill the missing data or remove the rows which have data missing.

Let's check out what a version of our `car_sales` `DataFrame` might look like with missing values.

In [61]:
```
# Option 1: Import car sales data with missing values from local file (stored on
car_sales_missing = pd.read_csv("../data/car-sales-missing-data.csv")

# Option 2: Import car sales data with missing values from GitHub (if the file is h
car_sales_missing = pd.read_csv("https://raw.githubusercontent.com/mrdbourke/zero
car_sales_missing
```

Out[61]:

	Make	Colour	Odometer	Doors	Price
0	Toyota	White	150043.0	4.0	$4,000
1	Honda	Red	87899.0	4.0	$5,000
2	Toyota	Blue	NaN	3.0	$7,000
3	BMW	Black	11179.0	5.0	$22,000
4	Nissan	White	213095.0	4.0	$3,500
5	Toyota	Green	NaN	4.0	$4,500
6	Honda	NaN	NaN	4.0	$7,500
7	Honda	Blue	NaN	4.0	NaN
8	Toyota	White	60000.0	NaN	NaN
9	NaN	White	31600.0	4.0	$9,700

Missing values are shown by `NaN` in pandas. This can be considered the equivalent of `None` in Python.

Let's use the `.fillna()` function to fill the `Odometer` column with the average of the other values in the same column.

In [62]:
```
# Fill Odometer column missing values with mean
car_sales_missing["Odometer"].fillna(car_sales_missing["Odometer"].mean(),
                                     inplace=False) # inplace is set to False by c
```

Out[62]:
```
0    150043.000000
1     87899.000000
2     92302.666667
3     11179.000000
4    213095.000000
5     92302.666667
6     92302.666667
7     92302.666667
8     60000.000000
9     31600.000000
Name: Odometer, dtype: float64
```

Now let's check the original `car_sales_missing` `DataFrame`.

In [63]:
```
car_sales_missing
```

Out[63]:

	Make	Colour	Odometer	Doors	Price
0	Toyota	White	150043.0	4.0	$4,000
1	Honda	Red	87899.0	4.0	$5,000
2	Toyota	Blue	NaN	3.0	$7,000
3	BMW	Black	11179.0	5.0	$22,000
4	Nissan	White	213095.0	4.0	$3,500
5	Toyota	Green	NaN	4.0	$4,500
6	Honda	NaN	NaN	4.0	$7,500
7	Honda	Blue	NaN	4.0	NaN
8	Toyota	White	60000.0	NaN	NaN

9	NaN	White	31600.0	4.0	$9,700

Because `inplace` is set to `False` (default), there's still missing values in the `"Odometer"` column.

Instead of using `inplace`, let's resassign the column to the filled version.

We'll use the syntax `df[col] = df[col].fillna(value)` to fill the missing values in the `"Odometer"` column with the average of the other values in the same column.

In [64]:
```
# Fill the Odometer missing values to the mean with inplace=True
car_sales_missing["Odometer"] = car_sales_missing["Odometer"].fillna(car_sales_mi
```

Now let's check the `car_sales_missing` `DataFrame` again.

In [65]:
```
car_sales_missing
```

Out[65]:

	Make	Colour	Odometer	Doors	Price
0	Toyota	White	150043.000000	4.0	$4,000
1	Honda	Red	87899.000000	4.0	$5,000
2	Toyota	Blue	92302.666667	3.0	$7,000
3	BMW	Black	11179.000000	5.0	$22,000
4	Nissan	White	213095.000000	4.0	$3,500
5	Toyota	Green	92302.666667	4.0	$4,500
6	Honda	NaN	92302.666667	4.0	$7,500
7	Honda	Blue	92302.666667	4.0	NaN
8	Toyota	White	60000.000000	NaN	NaN
9	NaN	White	31600.000000	4.0	$9,700

The missing values in the `Odometer` column have been filled with the mean value of the same column.

In practice, you might not want to fill a column's missing values with the mean, but this example was to show the difference between `inplace=False` (default) and `inplace=True`.

Whichever you choose to use will depend on how you structure your code.

All you have to remember is `inplace=False` returns a copy of the `DataFrame` you're working with.

This is helpful if you want to make a duplicate of your current `DataFrame` and save it to another variable.

Where as, `inplace=True` makes all the changes directly to the target `DataFrame`.

We've filled some values but there's still missing values in `car_sales_missing`. Let's say you wanted to remove any rows which had missing data and only work with rows which had complete coverage.

You can do this using `.dropna()`.

In [66]:
```
# Remove missing data
car_sales_missing.dropna()
```

Out[66]:

	Make	Colour	Odometer	Doors	Price
0	Toyota	White	150043.000000	4.0	$4,000
1	Honda	Red	87899.000000	4.0	$5,000
2	Toyota	Blue	92302.666667	3.0	$7,000
3	BMW	Black	11179.000000	5.0	$22,000
4	Nissan	White	213095.000000	4.0	$3,500
5	Toyota	Green	92302.666667	4.0	$4,500

It appears the rows with missing values have been removed, now let's check to make sure.

In [67]:
```
car_sales_missing
```

Out[67]:

	Make	Colour	Odometer	Doors	Price
0	Toyota	White	150043.000000	4.0	$4,000
1	Honda	Red	87899.000000	4.0	$5,000
2	Toyota	Blue	92302.666667	3.0	$7,000
3	BMW	Black	11179.000000	5.0	$22,000
4	Nissan	White	213095.000000	4.0	$3,500
5	Toyota	Green	92302.666667	4.0	$4,500
6	Honda	NaN	92302.666667	4.0	$7,500
7	Honda	Blue	92302.666667	4.0	NaN
8	Toyota	White	60000.000000	NaN	NaN
9	NaN	White	31600.000000	4.0	$9,700

Hmm, they're still there, can you guess why?

It's because `.dropna()` has `inplace=False` as default. We can either set `inplace=True` or reassign the `car_sales_missing` `DataFrame`.

In [68]:
```
# The following two lines do the same thing
car_sales_missing.dropna(inplace=True) # Operation happens inplace without reassig
car_sales_missing = car_sales_missing.dropna() # car_sales_missing gets reassigned
```

Now if check again, the rows with missing values are gone and the index numbers have been updated.

In [69]:
```
car_sales_missing
```

Out[69]:

	Make	Colour	Odometer	Doors	Price
0	Toyota	White	150043.000000	4.0	$4,000
1	Honda	Red	87899.000000	4.0	$5,000
2	Toyota	Blue	92302.666667	3.0	$7,000
3	BMW	Black	11179.000000	5.0	$22,000
4	Nissan	White	213095.000000	4.0	$3,500
5	Toyota	Green	92302.666667	4.0	$4,500

Instead of removing or filling data, what if you wanted to create it?

For example, creating a column called `Seats` for number of seats.

pandas allows for simple extra column creation on `DataFrame`'s.

Three common ways are:

1. Adding a `pandas.Series` as a column.
2. Adding a Python list as a column.
3. By using existing columns to create a new column.

In [70]:
```
# Create a column from a pandas Series
seats_column = pd.Series([5, 5, 5, 5, 5, 5, 5, 5, 5, 5])
car_sales["Seats"] = seats_column
car_sales
```

Out[70]:

	Make	Colour	Odometer (KM)	Doors	Price	Seats
0	toyota	White	150043	4	4000	5
1	honda	Red	87899	4	5000	5

2	toyota	Blue	32549	3	7000	5
3	bmw	Black	11179	5	22000	5
4	nissan	White	213095	4	3500	5
5	toyota	Green	99213	4	4500	5
6	honda	Blue	45698	4	7500	5
7	honda	Blue	54738	4	7000	5
8	toyota	White	60000	4	6250	5
9	nissan	White	31600	4	9700	5

Creating a column is similar to selecting a column, you pass the target `DataFrame` along with a new column name in brackets.

In [71]:
```
# Create a column from a Python list
engine_sizes = [1.3, 2.0, 3.0, 4.2, 1.6, 1, 2.0, 2.3, 2.0, 3.0]
car_sales["Engine Size"] = engine_sizes
car_sales
```

Out[71]:

	Make	Colour	Odometer (KM)	Doors	Price	Seats	Engine Size
0	toyota	White	150043	4	4000	5	1.3
1	honda	Red	87899	4	5000	5	2.0
2	toyota	Blue	32549	3	7000	5	3.0
3	bmw	Black	11179	5	22000	5	4.2
4	nissan	White	213095	4	3500	5	1.6
5	toyota	Green	99213	4	4500	5	1.0
6	honda	Blue	45698	4	7500	5	2.0
7	honda	Blue	54738	4	7000	5	2.3
8	toyota	White	60000	4	6250	5	2.0