Module 4: Fundamentals of Data Analysis

Quick Reference Guide

Learning Outcomes:

1. Use pandas.merge to join datasets
2. Create a variety of plot types in pandas, Seaborn, and Plotly
3. Discuss the advantages and disadvantages of each plotting library
4. Perform string manipulation in pandas
5. Import and clean messy data from real-world datasets

Basic GDP and Population Join

The pandas library can be used for more sophisticated manipulations of data.

For example, you can use a dataset that captures the GDP of various countries across the world to determine the relative per capita wealth of each country. This requires you to divide the GDP value by the population of a country.

In order to do so, there are several steps in the process.

1. Read-in an appropriate dataset (such as the World Bank dataset from an online source)
2. Observe the dataset for interesting details and/or trends (for example, whether the numbers are estimates)
3. Prepare the dataset for the per capita adjustments you want to make:

- Rename a column using this syntax: **pop.rename(columns = {"Old column name" : "New column name"})**
- Reassign the updated population variable to the new table
- Specify a particular year and population to assess, for instance populations in 2017:

For the GDP in 2017, use: **gdp2017 = gdp.query("Year == 2017")**

	Entity	Code	Year	GDP (constant 2010 US$)	gdp
15	Afghanistan	AFG	2017	2.196941e+10	21.969414
53	Albania	ALB	2017	1.398856e+10	13.988556
111	Algeria	DZA	2017	1.993674e+11	199.367414
174	Andorra	AND	2017	3.382068e+09	3.382068
212	Angola	AGO	2017	1.037860e+11	103.785984
...	...	...	...	...	...
8597	Uzbekistan	UZB	2017	6.577995e+10	65.779950
8636	Vanuatu	VUT	2017	8.120250e+08	0.812025
8725	Vietnam	VNM	2017	1.752841e+11	175.284081
8810	Zambia	ZMB	2017	2.813940e+10	28.139397
8868	Zimbabwe	ZWE	2017	1.532981e+10	15.329811

For the 2017 population, use: **pop2017 = pop.query("Year == 2017")**

	Entity	Code	Year	population
217	Afghanistan	AFG	2017	36296000
339	Africa	NaN	2017	1244221952
559	Albania	ALB	2017	2884000
779	Algeria	DZA	2017	41389000
849	American Samoa	ASM	2017	56000
...	...	...	...	...
46069	Western Sahara	ESH	2017	553000
46220	World	OWID_WRL	2017	7547858944
46440	Yemen	YEM	2017	27835000
46660	Zambia	ZMB	2017	16854000
46880	Zimbabwe	ZWE	2017	14237000

Join the GDP and population tables for the year 2017 with the **pandas.merge()** function:

- Enumerate which tables you wish to join, for example: **gdp_and pop_2017 = pd.merge()**
- Specify how you want to join the data, for example: **gdp_and pop2017 = pd.merge (left = gdp2017, right = pop2017)**
- Line up rows based on the entity value by adding **left_on = "Entity"** and **right_on = "Entity"**
- Use **how = "outer"** to add a parameter (for instance, an outer join)

	Entity	Code_x	Year_x	GDP (constant 2010 US$)	gdp	Code_y	Year_y	population
0	Afghanistan	AFG	2017.0	2.196941e+10	21.969414	AFG	2017.0	3.629600e+07
1	Albania	ALB	2017.0	1.398856e+10	13.988556	ALB	2017.0	2.884000e+06
2	Algeria	DZA	2017.0	1.993674e+11	199.367414	DZA	2017.0	4.138900e+07
3	Andorra	AND	2017.0	3.382068e+09	3.382068	AND	2017.0	7.700000e+04
4	Angola	AGO	2017.0	1.037860e+11	103.785984	AGO	2017.0	2.981700e+07
...	...	...	...	...	...	...	...	...
238	Venezuela	NaN	NaN	NaN	NaN	VEN	2017.0	2.940200e+07
239	Wallis and Futuna	NaN	NaN	NaN	NaN	WLF	2017.0	1.200000e+04
240	Western Sahara	NaN	NaN	NaN	NaN	ESH	2017.0	5.530000e+05
241	World	NaN	NaN	NaN	NaN	OWID_WRL	2017.0	7.547859e+09
242	Yemen	NaN	NaN	NaN	NaN	YEM	2017.0	2.783500e+07

Calculate the GDP per capita value for the joined table:
gdp_and_pop_2017["gdp per capita"] = gdp_and_pop_2017["GDP (constant 2010 US$)"] / gdp_and_pop_2017 ["population"]
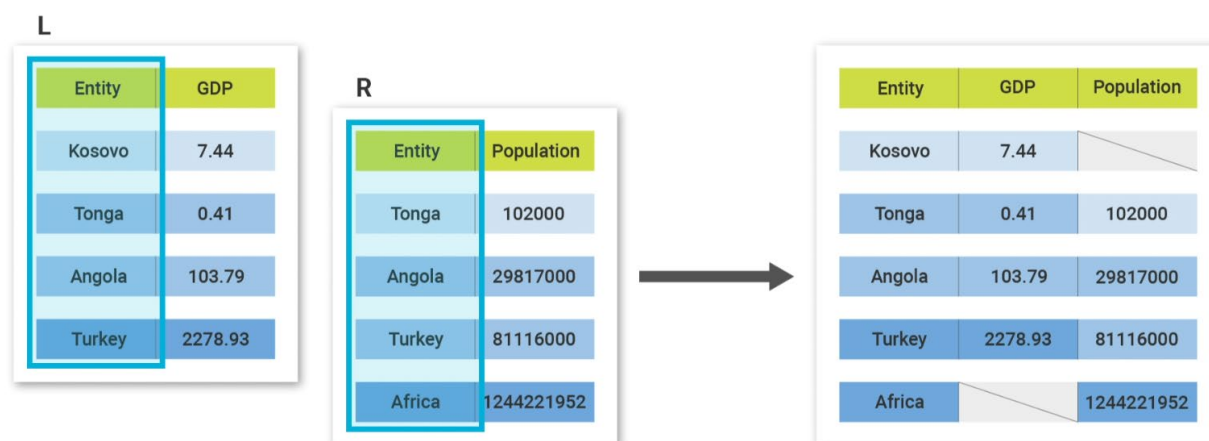
	Entity	Code_x	Year_x	GDP (constant 2010 US$)	gdp	Code_y	Year_y	population	gdp_per_capita
0	Afghanistan	AFG	2017.0	2.196941e+10	21.969414	AFG	2017.0	3.629600e+07	605.284718
1	Albania	ALB	2017.0	1.398856e+10	13.988556	ALB	2017.0	2.884000e+06	4850.400743
2	Algeria	DZA	2017.0	1.993674e+11	199.367414	DZA	2017.0	4.138900e+07	4816.917876
3	Andorra	AND	2017.0	3.382068e+09	3.382068	AND	2017.0	7.700000e+04	43922.964124
4	Angola	AGO	2017.0	1.037860e+11	103.785984	AGO	2017.0	2.981700e+07	3480.765485
...	...	...	...	...	...	...	...	...	...
238	Venezuela	NaN	NaN	NaN	NaN	VEN	2017.0	2.940200e+07	NaN
239	Wallis and Futuna	NaN	NaN	NaN	NaN	WLF	2017.0	1.200000e+04	NaN
240	Western Sahara	NaN	NaN	NaN	NaN	ESH	2017.0	5.530000e+05	NaN
241	World	NaN	NaN	NaN	NaN	OWID_WRL	2017.0	7.547859e+09	NaN
242	Yemen	NaN	NaN	NaN	NaN	YEM	2017.0	2.783500e+07	NaN

More Basic Joins with a Diagram

The **pd.merge()** function allows for different join types through the use of the how parameter. The types include **left**, **right**, **inner**, and **outer**. For instance, you can use **how = "outer"** to create an outer join or **how = "left"** to create a left join.

Choosing a specific type of join is important when it comes to columns which may be present in only one of the tables you are joining, as well as determining the importance of the table data in the join.

For example, if an entity is present in one table but not in another, using an **outer join** means the entity is included (with some values missing) in the joined table even if the original table columns did not include data on certain aspects.

L	
Entity	**GDP**
Kosovo	7.44
Tonga	0.41
Angola	103.79
Turkey	2278.93

R	
Entity	**Population**
Tonga	102000
Angola	29817000
Turkey	81116000
Africa	1244221952

Entity	GDP	Population
Kosovo	7.44	
Tonga	0.41	102000
Angola	103.79	29817000
Turkey	2278.93	81116000
Africa		1244221952

If one entity in a table is considered more important when merging with another table, a **left** or **right join** should be used – **left** if the more important table is on the left, and **right** if the more important table is on the right.

A right join is essentially the opposite of a left join.

Shown here is the result of **how = "left"** in an example:

If you use an **inner join**, pandas keeps the rows where the entity in question is present in both tables.

However, be aware that if non-unique values exist in both tables, the resulting join may display all the possible combinations of the matching rows.

Joining by Multiple Fields

The ability to **join multiple fields using multiple variables** is a useful tool in pandas.

For example, you may want to view the **GDP per capita of multiple countries for multiple years**. In order to do so, you can merge the entire GDP and population tables.

However, to avoid getting an output of duplicated entity values (which shows all possible combinations in the joined table), you need to refine the syntax.

You can do this in a way that joins on the entity as well as the year.

For example, to join the Afghanistan GDP values and population values for the year 2002, you can use this syntax:

gdp_and_pop = pd.merge(left = gdp, right = pop, left_on = ["Entity", "Year"], right_on = ["Entity", "Year"], how = "left"

	Entity	Code_x	Year	GDP (constant 2010 US$)	gdp	Code_y	population
0	Afghanistan	AFG	2002	8.013233e+09	8.013233	AFG	22601000.0
1	Afghanistan	AFG	2003	8.689884e+09	8.689884	AFG	23681000.0
2	Afghanistan	AFG	2004	8.781610e+09	8.781610	AFG	24727000.0
3	Afghanistan	AFG	2005	9.762979e+09	9.762979	AFG	25654000.0
4	Afghanistan	AFG	2006	1.030523e+10	10.305228	AFG	26433000.0
...	...	...	...	...	...	...	...
8864	Zimbabwe	ZWE	2013	1.418193e+10	14.181927	ZWE	13350000.0
8865	Zimbabwe	ZWE	2014	1.448359e+10	14.483588	ZWE	13587000.0
8866	Zimbabwe	ZWE	2015	1.472830e+10	14.728302	ZWE	13815000.0
8867	Zimbabwe	ZWE	2016	1.481899e+10	14.818986	ZWE	14030000.0
8868	Zimbabwe	ZWE	2017	1.532981e+10	15.329811	ZWE	14237000.0

To view the GDP per capita values, you divide the GDP by the population:

**gdp_and_pop["gdp per capita"] = gdp_and_pop["GDP (constant 2010 US$)"] / gdp_and_pop["population"]**

	Entity	Code_x	Year	GDP (constant 2010 US$)	gdp	Code_y	population	gdp_per_capita
0	Afghanistan	AFG	2002	8.013233e+09	8.013233	AFG	22601000.0	354.552149
1	Afghanistan	AFG	2003	8.689884e+09	8.689884	AFG	23681000.0	366.955940
2	Afghanistan	AFG	2004	8.781610e+09	8.781610	AFG	24727000.0	355.142564
3	Afghanistan	AFG	2005	9.762979e+09	9.762979	AFG	25654000.0	380.563610
4	Afghanistan	AFG	2006	1.030523e+10	10.305228	AFG	26433000.0	389.862222
...	...	...	...	...	...	...	...	...
8864	Zimbabwe	ZWE	2013	1.418193e+10	14.181927	ZWE	13350000.0	1062.316603
8865	Zimbabwe	ZWE	2014	1.448359e+10	14.483588	ZWE	13587000.0	1065.988675
8866	Zimbabwe	ZWE	2015	1.472830e+10	14.728302	ZWE	13815000.0	1066.109450
8867	Zimbabwe	ZWE	2016	1.481899e+10	14.818986	ZWE	14030000.0	1056.235654
8868	Zimbabwe	ZWE	2017	1.532981e+10	15.329811	ZWE	14237000.0	1076.758501

8869 rows × 8 columns

Next, you can plot and view the GDP per capita of countries over time.

For an individual country, such as Norway, you use this syntax:
px.line(gdp_and_pop.query('Entity == "Norway"'), x = "Year, y = "gdp_per_capita")
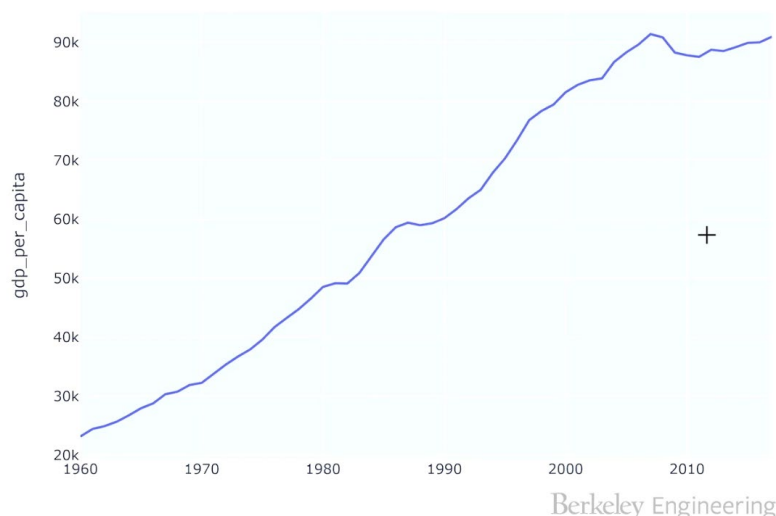
For all the countries in the dataset, you can use this syntax:
px.line(gdp_and_pop, x = "Year", y = "gdp_per_capita", color = "Entity")
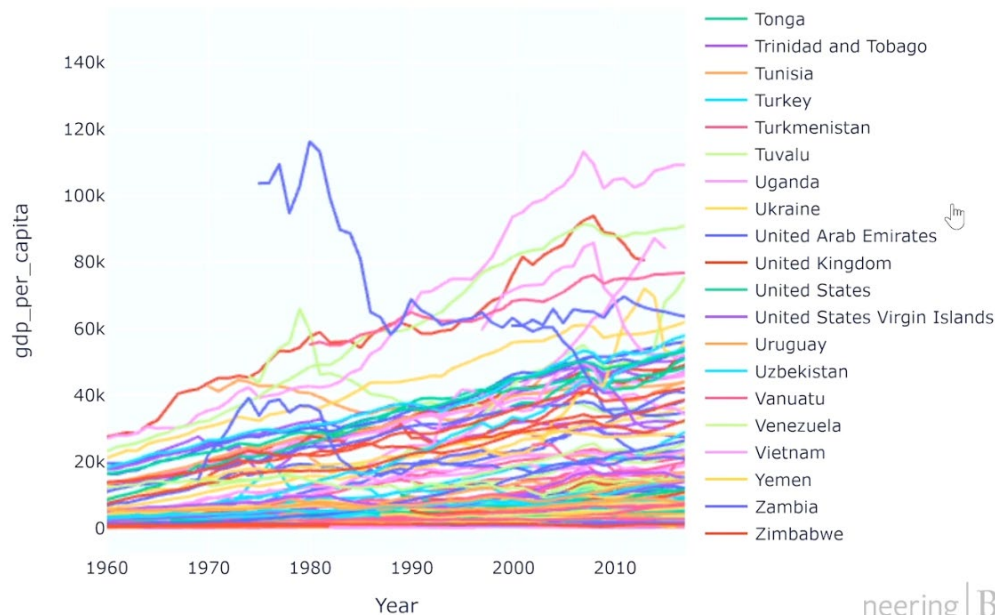
You can also view **how much wealthier, on a per capita basis, a country is now compared to the past**, using the normalized GDP figure. To do this, you will create a temporary dataframe to use as a denominator.

For instance, if you wish to create a plot to view how the wealth of various countries have grown since 1960, you can follow this process:

1. Set the entity as the table's index: **gdp_and_pop_by_entity = gdp_and_pop.set_index("Entity")**

Create a temporary 1960 dataframe: **gdp_per_capitas_1960 = gdp_and_pop_by_entity.query("Year" == 1960)[["gdp_per_capita"]]**
Divide the indexed table by the dataframe:
gdp_and_pop_by_entity["gdp_per_cap_ratio"] = gdp_and_pop_by_entity["gdp_per_capita"] / gdp_per_capitas_1960["gdp_per_capita"]

Reset the index: **gdp_and_pop = gdp_and_pop_by_entity.reset_index()**

Drop the null rows (to exclude variables that have no value):
gdp_per_cap_ratio_history = gdp_and_pop.dropna()

Create the plot: **px.line(gdp_per_cap_ratio_history, x = "Year", y = "gdp_per_cap_ratio", color = "Entity")**

Scatterplot of GDP and Population

The development of a country can be visualized through the use of various ratios.

- **GDP ratio**: The economic output of a country as compared to its output in a specified time (for example, during the year 1960)
- **GDP per capita ratio**: The population adjusted GDP as compared with the GDP in a specified time (now versus 1960, for instance)
- **Population ratio**: How many more people inhabit a country now than did so in a specified time (for example, now versus 1960)

These ratios can be visualized using a scatterplot, which allows you to see a number of ratios at the same time.

For example, if you wish to view all three of the ratios—GDP, GDP per capita, and population ratio—for only one year, you can specify that in the syntax.

Here is the syntax for a scatterplot comparing 2017 ratios:
px.scatter(gdp_and_pop.query("Year == 2017"), x = "gdp_per_cap_ratio", y = "pop_ratio", color = "Entity", size = "gdp_ratio")
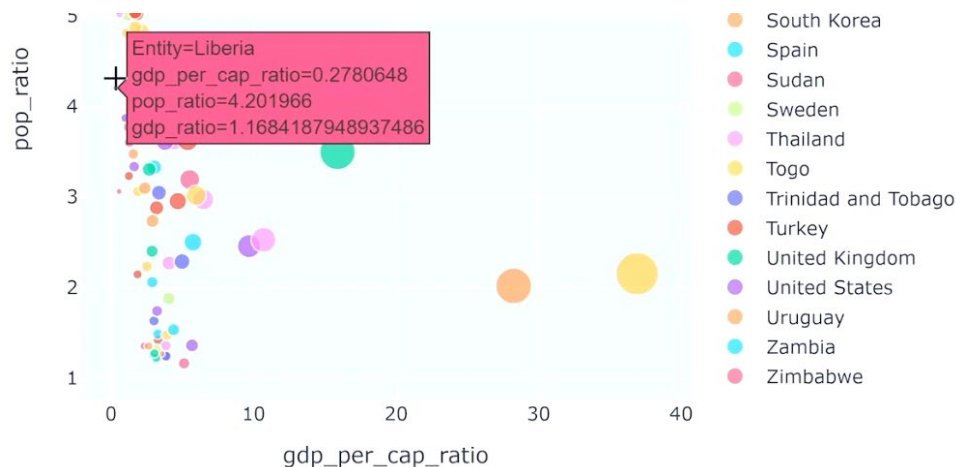
This lets you view all the ratios for the country, with the size of each dot on the scatterplot indicating the relative GDP growth of a country since 1960. If you hover over a country, you can diagnose the cause of the GDP growth by viewing the individual ratio data for that country on the scatterplot.

Incorporating Life Expectancy Data

If you already have a joined table—such as the GDP data and population table—you can continue to add measures, for example life expectancy.

To do this, you first read-in life expectancy data from your online source:
life_exp = pd.read_csv("life_expectancy.csv")

	Entity	Code	Year	Life expectancy
0	Afghanistan	AFG	1950	27.638
1	Afghanistan	AFG	1951	27.878
2	Afghanistan	AFG	1952	28.361
3	Afghanistan	AFG	1953	28.852
4	Afghanistan	AFG	1954	29.350
...	...	...	...	...
18440	Zimbabwe	ZWE	2015	59.534
18441	Zimbabwe	ZWE	2016	60.294
18442	Zimbabwe	ZWE	2017	60.812
18443	Zimbabwe	ZWE	2018	61.195
18444	Zimbabwe	ZWE	2019	61.490

18445 rows × 4 columns

If you remove the Code column, this leaves only the Entity Year and Life expectancy columns. To do this, you use this syntax:

life_exp = life.exp[["Entity", "Year", "Life expectancy"]]

You can join the life expectancy table to the created GDP and population table with this syntax:

gdp_pop_le = pd.merge(left = gdp_and_pop, right = life_exp, left_on = ["Entity", "Year"], right_on = ["Entity", "Year"], how = "left")

Entity	Code_x	Year	GDP (constant 2010 US$)	gdp	Code_y	population	gdp_per_capita	gdp_per_cap_ratio	pop_ratio	gdp_ratio	Life expectancy
Algeria	DZA	1960	2.743440e+10	27.434399	DZA	11058000.0	2480.954892	1.000000	1.000000	1.000000	46.141
Algeria	DZA	1961	2.370183e+10	23.701828	DZA	11336000.0	2090.845811	0.842758	1.025140	0.863946	46.599
Algeria	DZA	1962	1.903611e+10	19.036113	DZA	11620000.0	1638.219736	0.660318	1.050823	0.693878	47.056
Algeria	DZA	1963	2.556811e+10	25.568114	DZA	11913000.0	2146.236350	0.865085	1.077320	0.931973	47.509
Algeria	DZA	1964	2.706114e+10	27.061141	DZA	12222000.0	2214.133643	0.892452	1.105263	0.986395	47.958
...	...	...	...	...	...	...	...	...	...	...	...
Zimbabwe	ZWE	2013	1.418193e+10	14.181927	ZWE	13350000.0	1062.316603	1.192968	3.534551	4.216605	56.897
Zimbabwe	ZWE	2014	1.448359e+10	14.483588	ZWE	13587000.0	1065.988675	1.197091	3.597299	4.306296	58.410
Zimbabwe	ZWE	2015	1.472830e+10	14.728302	ZWE	13815000.0	1066.109450	1.197227	3.657665	4.379055	59.534

The output has many columns and is an example of **high-dimensional data** — which is typical of large data analysis projects.

Once the data is merged, you can create a scatterplot to visualize the implicit relationship between GDP and life expectancy. Use this syntax:

**px.scatter(gdp_pop_le.query("Year == 2017"), x = "gdp_per_capita", y = "Life expectancy"), color = "Entity")**
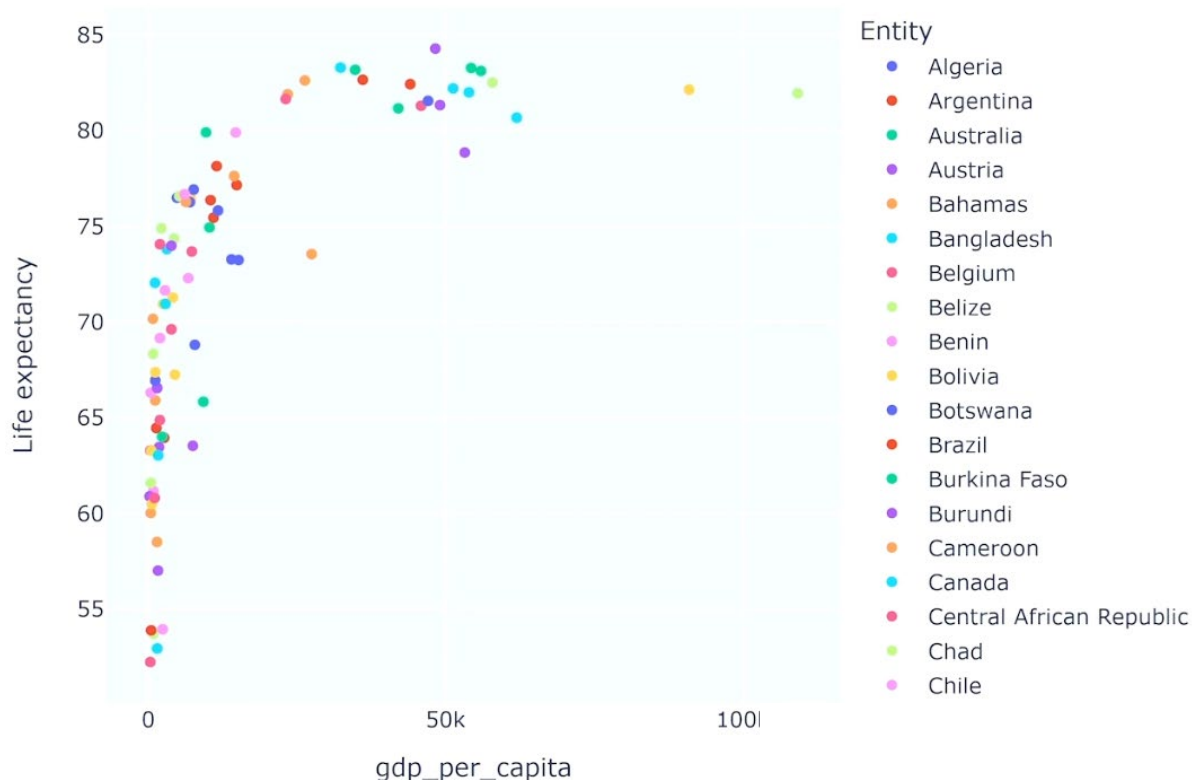
The scatterplot indicates a trend that suggests life expectancy is generally increased in countries with a higher GDP.

Since the scatterplot shows a curve, you might want to plot the data on an axis where the x is logarithmic, using the **logarithmic x-axis feature of Plotly**.

You can achieve this by adding **log_x = True** to your line of code.

The shape of the resulting logarithmic scatterplot changes to display a more linear relationship, which suggests that perhaps a machine learning model can be built to predict life expectancy based on a country's GDP.
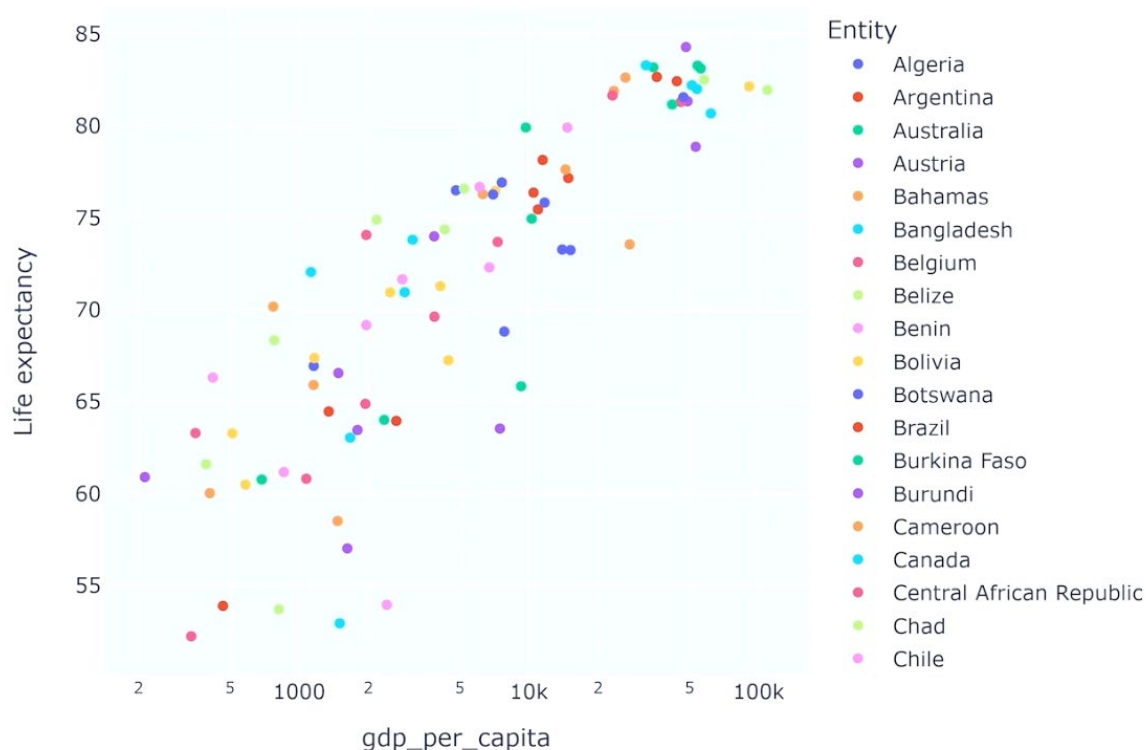
It is clear that the combination and manipulation of datasets opens up all kinds of questions for further analysis.

Violin and Box Plots

Data can be presented in various ways, including classic plots, bar plots, scatterplots, violin plots, box plots, and histograms using either Plotly or Seaborn.

For example, you can create a simple **Plotly histogram** to represent GDP and life expectancy data using this syntax:

px.histogram(gdp_pop_le.query("Year == 2017")["Life expectancy"])
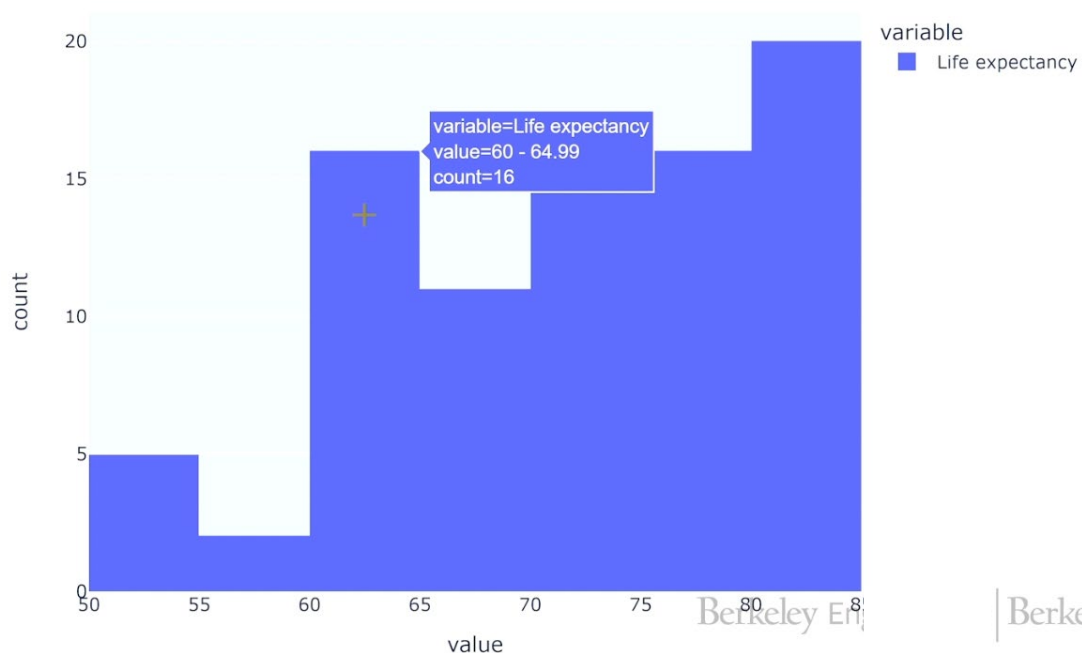
Plotly histograms are interactive in the sense that you can get information by hovering your mouse over any part of the diagram to view details. The bins in Plotly histograms are not outlined by default.

You can also create a **Seaborn histogram** to represent the GDP and life expectancy data with this syntax:

sns.displot(gdp_pop_le.query("Year == 2017")["Life expectancy"])
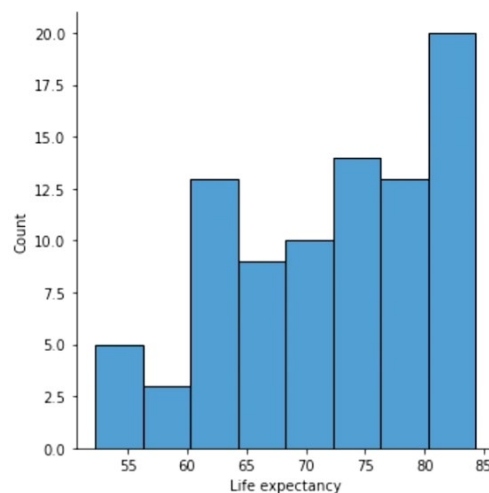
Seaborn histograms are not interactive, but have outlined bins and the data is represented clearly. They support additional features for a better understanding of the distribution of data. For instance, if you add **kde = True** to the code, it overlays the **kernel density estimate (KDE)** on the histogram. The KDE overlay gives a smoothed version of the histogram to show the rough distribution of the life expectancy data.
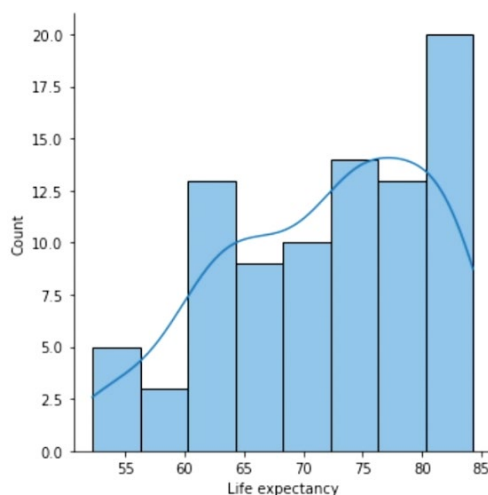
With Seaborn, you can create a **rug plot** by adding **rug = True** to the code:

sns.displot(gdp_pop_le.query("Year == 2017")["Life_expectancy"], kde = True, rug = True)
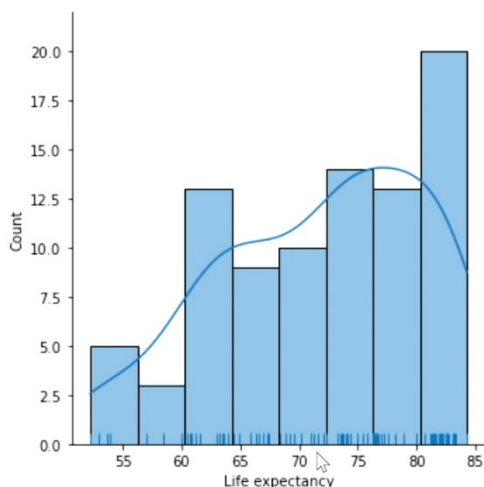
This overlays every single datapoint from the original dataset on the histogram's x-axis.

You can create **violin plots** in Plotly, which give a sideways version of the KDE curve to show the distribution of life expectancies in different countries.

To add datapoints to a violin plot, add **points = "all"** to the code:

px.violin(gdp_pop_le.query("Year == 2017")["Life_expectancy"], points = "all")
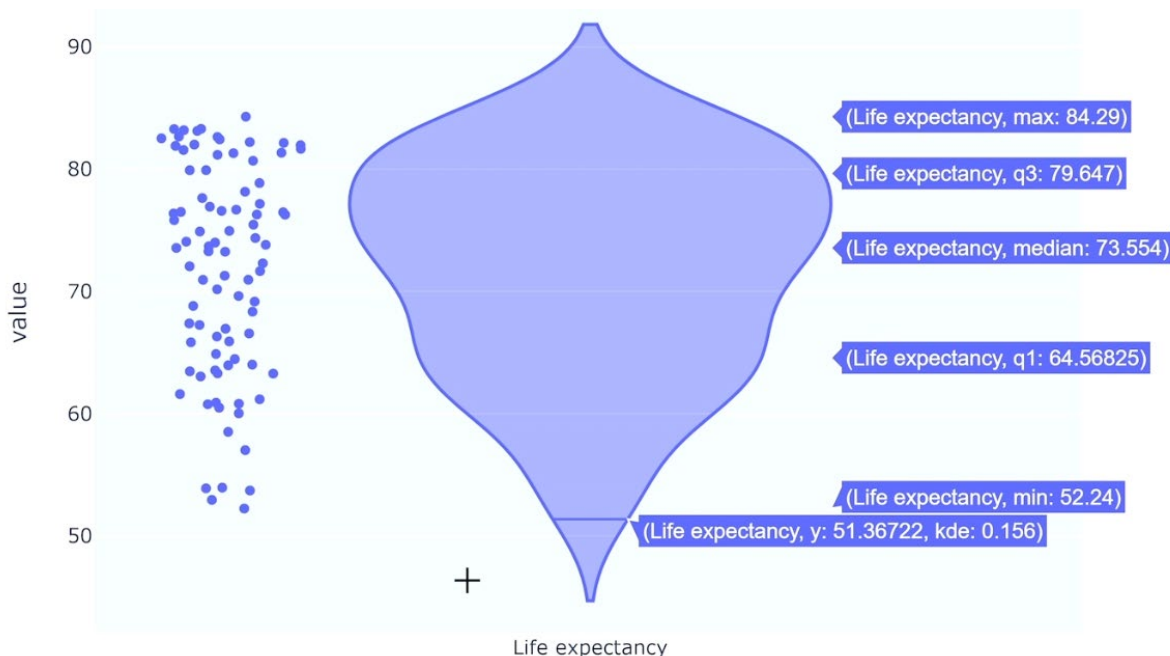
Violin plots are useful if you want to further augment the data, for example when segmented by different categories.

So, to start to segment and compare the data using a Plotly violin plot, you can read-in another category, such as continents:

continents = pd.read_csv("continents.csv")

You can then merge the data, as before:

gdp_pop_le = pd.merge(left = gdp_pop_le, right = continents, left_on = "Entity", right_on = "Country", how = "left")

Once you have the continent data merged, you can use a violin plot to compare the life expectancies of two different continents, for instance Africa and Europe:

continents = ["Africa", "Europe"]

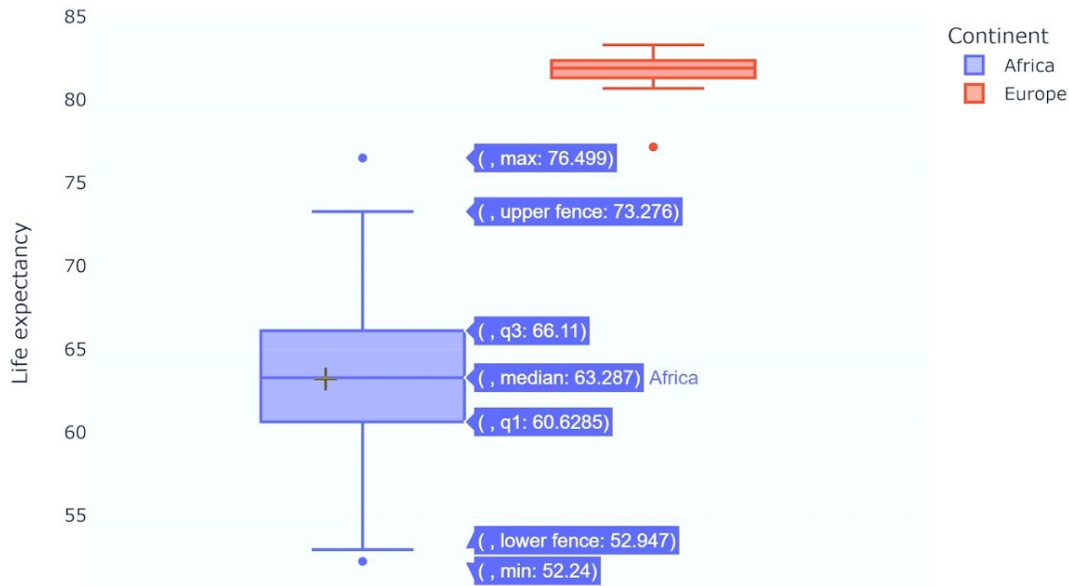**px.violin(gdp_pop_le.query("Year == 2017 and Continent in @continents").dropna(), y = "Life expectancy", color = "Continent")**
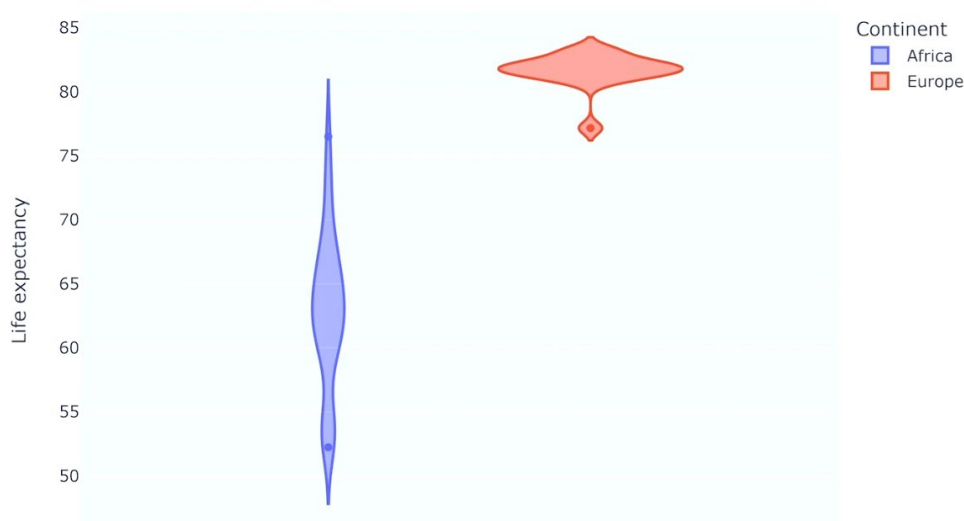
Box plots can be used in Plotly to display all the data as raw boxes. Box plots indicate data distribution through their medians, quartiles, and fences.

The box plot syntax is similar to that of a violin plot:

px.box(gdp_pop_le.query("Year == 2017 and Continent in @continents").dropna(), y = "Life expectancy", color = "Continent")

Since box plots neatly divide data, you can use them to visualize the distribution of different categories. For instance, to visualize the life expectancy data for all continents you would use this syntax:

**px.box(gdp_pop_le.query("Year == 2017").dropna(), y = "Life expectancy", color = "Continent")**
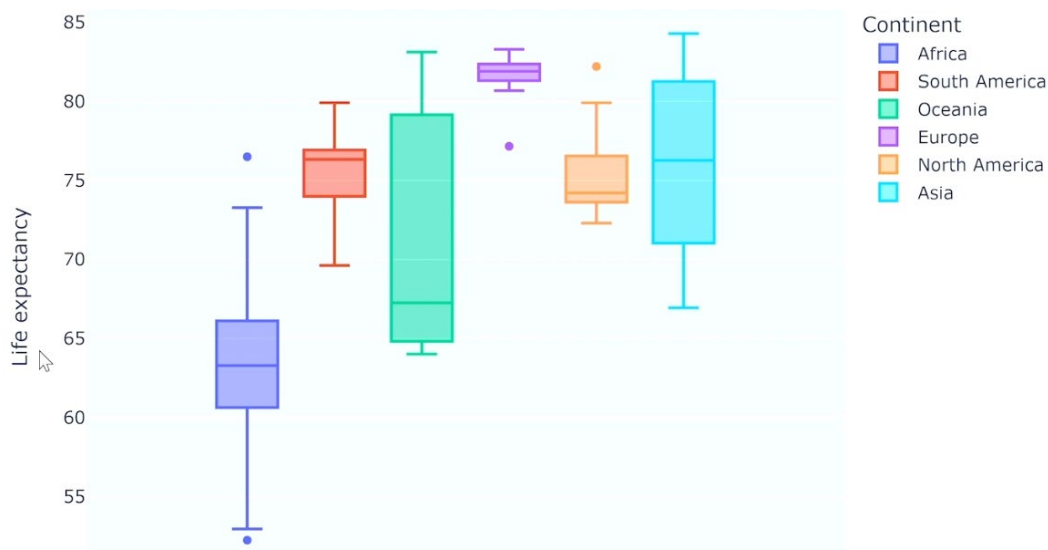
Joint Plots

Seaborn and Plotly can be used to join and create various types of plots and maps. The scatterplot function in Plotly allows you to create a **marginal plot** on the y and the x-axis. This means that the **marginal distribution** of each of the variables is displayed along with the scatterplot.

This type of joint plot allows for certain salient features to be clearly shown, allowing you to compile concise reports when discussing the results of a data analysis.

For example, a marginal box plot can be added to a scatterplot like this:

px.scatter(gdp_pop_le.query("Year == 2017"), x = "gdp_per_capita", y = "Life expectancy", marginal_y = "box", marginal_x = "box")
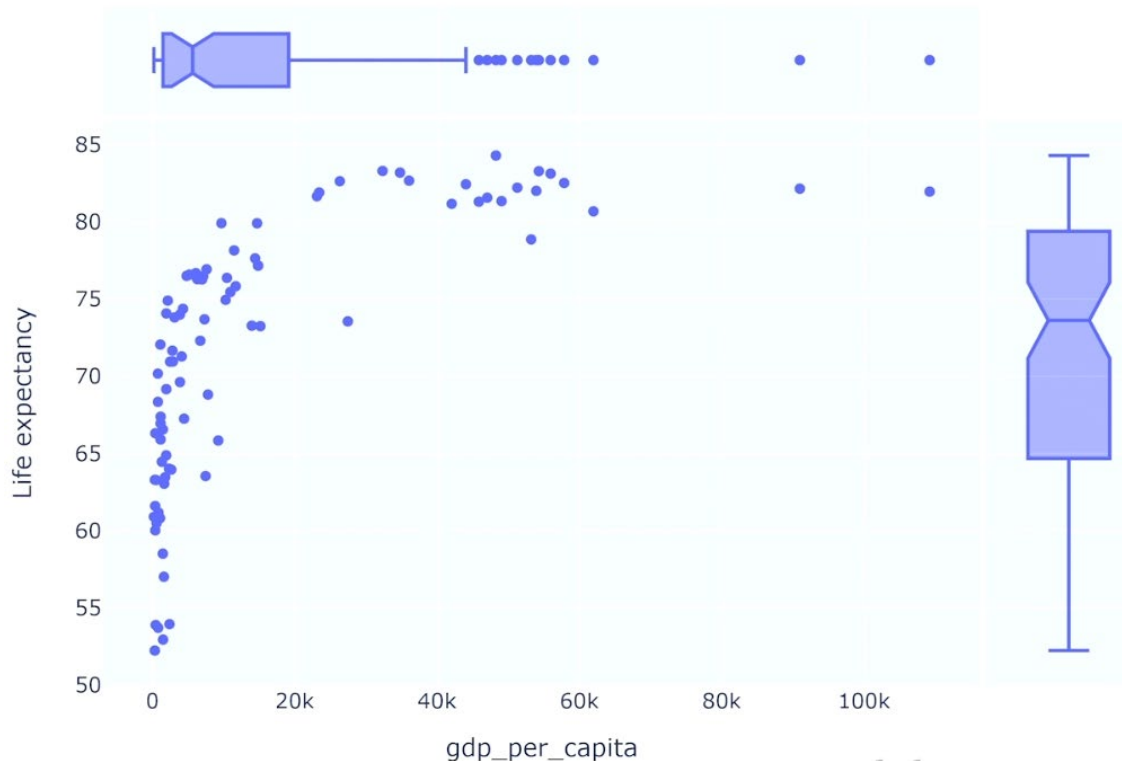
If you wish to capture the density of datapoints, you can use something in Plotly called a **density heatmap plot**, which draws attention to the overall distribution of datapoints.

For example, when used with a histogram on the same data in Plotly, the interactive plot indicates some relative data patterns with colors:

px.density_heatmap(gdp_pop_le.query("Year == 2017"), x = "gdp_per_capita", y = "Life expectancy", marginal_x = "histogram", marginal_y = "histogram")
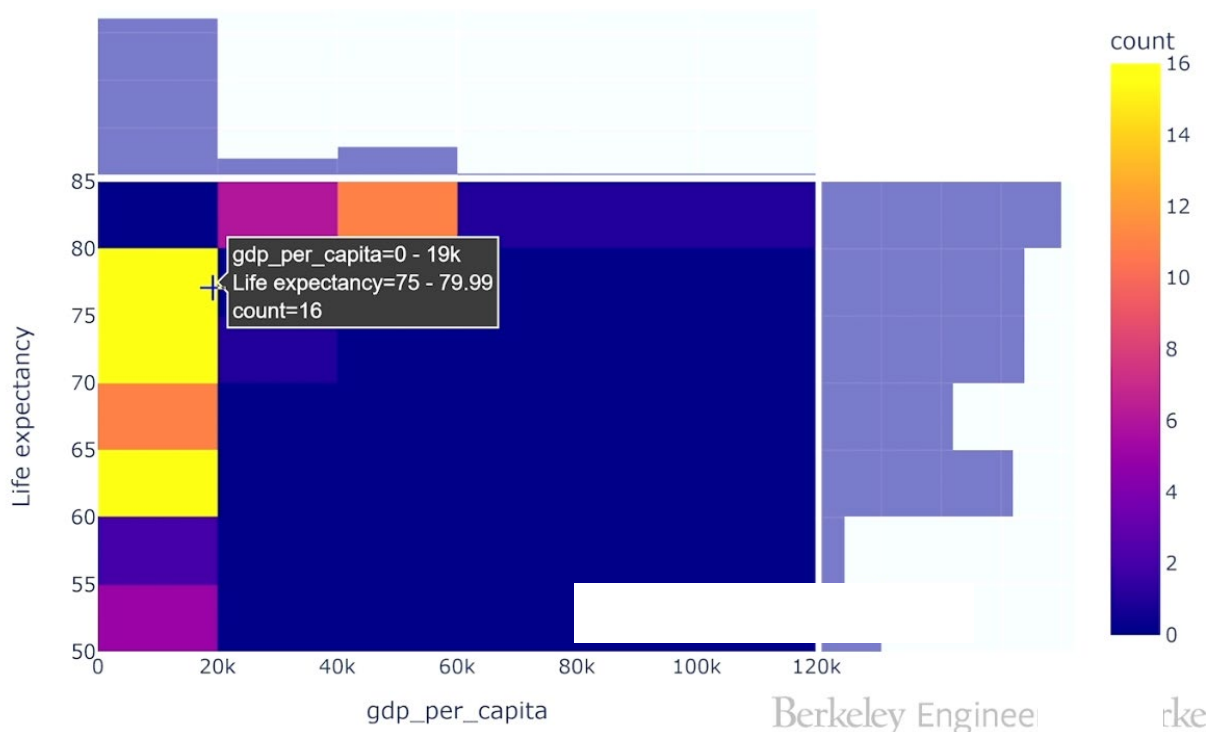
Seaborn also supports marginal plots and has its own jointplot function called hex. This displays the marginal data in hexagons along the x and y-axes in the plots:

sns.jointplot(data = gdp_pop_le.query("Year == 2017"), x = "gdp_per_capita", y = "Life expectancy", kind = "hex")
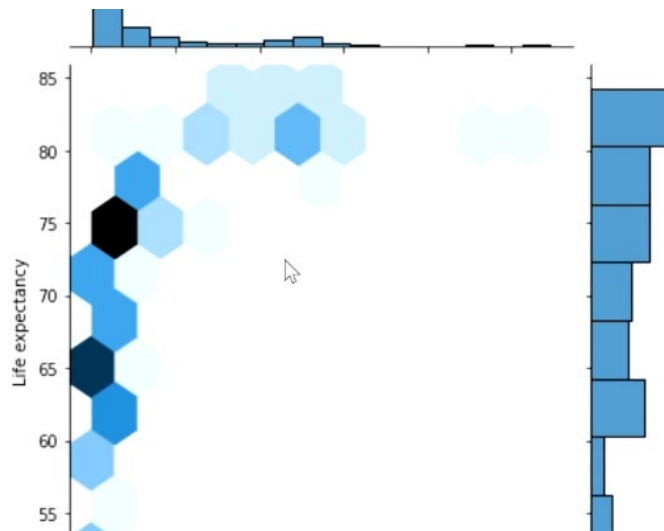
String Operations and Data Cleaning

The ability to manipulate strings is a useful tool in pandas, especially when working with large string libraries. In pandas there are various string functions, including **.str.startswith()**, **.str.contains()**, **.str.swapcase()**, **.str.strip()**, **.str.rsplit()**, **.str.startswith()**, and **.str.replace()**. An important function to use is **.str.contains()**, which is actually a library of vectorized string functions.

If you wish to search a dataset for only the countries whose name contain the substring "in", for instance, you can do so:

gdp_pop_le[gdp_pop_le["Entity"].str.contains("in")]

	Entity	Code_x	Year	GDP (constant 2010 US$)	gdp	Code_y	population	gdp_per_capita	gdp_per_cap_ratio	pop_ra
58	Argentina	ARG	1960	1.155739e+11	115.573869	ARG	20482000.0	5642.704253	1.000000	1.000(
59	Argentina	ARG	1961	1.218470e+11	121.847037	ARG	20817000.0	5853.246697	1.037312	1.016;
60	Argentina	ARG	1962	1.208089e+11	120.808874	ARG	21153000.0	5711.193378	1.012138	1.032
61	Argentina	ARG	1963	1.143961e+11	114.396101	ARG	21489000.0	5323.472508	0.943426	1.049
62	Argentina	ARG	1964	1.259848e+11	125.984766	ARG	21824000.0	5772.762383	1.023049	1.065;
...	...	...	...	...	...	...	...	...	...	
4978	United Kingdom	GBR	2013	2.564905e+12	2564.904713	GBR	64984000.0	39469.788151	2.852872	1.240;
4979	United Kingdom	GBR	2014	2.643243e+12	2643.243341	GBR	65423000.0	40402.356073	2.920278	1.249;

Another important function is **.str.startswith()**, which you can use to search the library for strings starting with a certain letter or letters. For example, if you enter **.str.startswith("B")**, the output will show countries starting with the letter B alphabetically listed.

You can also manipulate the dataset, for instance if you wish to uppercase all the entities in your table; you can use the **.str.upper()** function to do this.

These types of functions are really useful when joining tables from different sources, which may appear in different cases.

Real-World High-Dimensional Data

To practice working with real-world high-dimensional datasets, you can use data from platforms like **Kaggle** — a platform for data science and machine learning. The ability to predict sale prices is very practical and useful in the real estate market.

So, to consider how the sale price of a house varies as a function of other properties, for instance, you can use a large housing dataset table where each row represents a sale. First, you read it in:

df = pd.read_csv("housing.csv")

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities	...	PoolArea
0	1	60	RL	65.0	8450	Pave	NaN	Reg	Lvl	AllPub	...	0
1	2	20	RL	80.0	9600	Pave	NaN	Reg	Lvl	AllPub	...	0
2	3	60	RL	68.0	11250	Pave	NaN	IR1	Lvl	AllPub	...	0
3	4	70	RL	60.0	9550	Pave	NaN	IR1	Lvl	AllPub	...	0
4	5	60	RL	84.0	14260	Pave	NaN	IR1	Lvl	AllPub	...	0
...	...	...	...	...	...	...	...	...	...	...	...	...
1455	1456	60	RL	62.0	7917	Pave	NaN	Reg	Lvl	AllPub	...	0
1456	1457	20	RL	85.0	13175	Pave	NaN	Reg	Lvl	AllPub	...	0
1457	1458	70	RL	66.0	9042	Pave	NaN	Reg	Lvl	AllPub	...	0
1458	1459	20	RL	68.0	9717	Pave	NaN	Reg	Lvl	AllPub	...	0
1459	1460	20	RL	75.0	9937	Pave	NaN	Reg	Lvl	AllPub	...	0
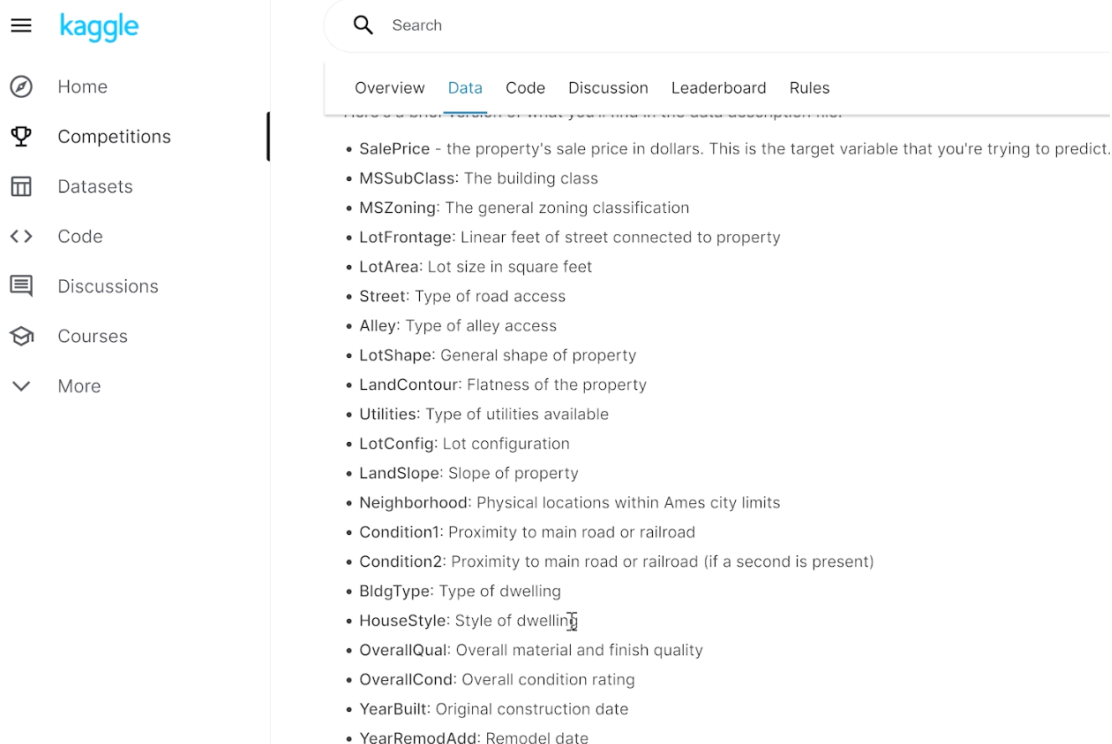
1460 rows × 81 columns

If it is a very large table with many columns, you can more easily view the columns by using the **df.columns()** function:

```
Index(['Id', 'MSSubClass', 'MSZoning', 'LotFrontage', 'LotArea', 'Street',
       'Alley', 'LotShape', 'LandContour', 'Utilities', 'LotConfig',
       'LandSlope', 'Neighborhood', 'Condition1', 'Condition2', 'BldgType',
       'HouseStyle', 'OverallQual', 'OverallCond', 'YearBuilt', 'YearRemodAdd',
       'RoofStyle', 'RoofMatl', 'Exterior1st', 'Exterior2nd', 'MasVnrType',
       'MasVnrArea', 'ExterQual', 'ExterCond', 'Foundation', 'BsmtQual',
       'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinSF1',
       'BsmtFinType2', 'BsmtFinSF2', 'BsmtUnfSF', 'TotalBsmtSF', 'Heating',
       'HeatingQC', 'CentralAir', 'Electrical', '1stFlrSF', '2ndFlrSF',
       'LowQualFinSF', 'GrLivArea', 'BsmtFullBath', 'BsmtHalfBath', 'FullBath',
       'HalfBath', 'BedroomAbvGr', 'KitchenAbvGr', 'KitchenQual',
       'TotRmsAbvGrd', 'Functional', 'Fireplaces', 'FireplaceQu', 'GarageType',
       'GarageYrBlt', 'GarageFinish', 'GarageCars', 'GarageArea', 'GarageQual',
       'GarageCond', 'PavedDrive', 'WoodDeckSF', 'OpenPorchSF',
       'EnclosedPorch', '3SsnPorch', 'ScreenPorch', 'PoolArea', 'PoolQC',
       'Fence', 'MiscFeature', 'MiscVal', 'MoSold', 'YrSold', 'SaleType',
       'SaleCondition', 'SalePrice'],
      dtype='object')
```

Since column names are not always self-explanatory, you can get a description of the data from the Kaggle website's **data_description.txt** file.

You need to explore the real-world dataset to understand its contents, the various categorical values, and how often they appear in the dataset.

For instance, you may want to view the street feature on its own to ascertain which values are displayed often. This can help determine whether the figures will be useful in a model for predicting housing sale prices. So you run the function:

df["Street"]

In this example, you determine that the street feature is not useful, since basically all of the "Street" rows contain Pave.

```
0        Pave
1        Pave
2        Pave
3        Pave
4        Pave
        ...
1455     Pave
1456     Pave
1457     Pave
1458     Pave
1459     Pave
Name: Street, Length: 1460, dtype: object
```

To summarize how often a categorical value appears in the dataset, you can use the **value_counts()** function:

df["Street"], value_counts()

```
: Pave     1454
  Grvl        6
  Name: Street, dtype: int64
```

Suppose you want to try and show the sale price as a function of some of the other features.

For real estate, the most intuitive features to look at are those associated with square footage, which means you need to look at the columns that relate to the area the house occupies.

To see the columns with applicable information, you can go to the Kaggle website's data description document and search for columns containing the word "area", or you can do it by using the **.str.contains()** function:

df.columns[df.columns.str.contains("Area")]

```
Index(['LotArea', 'MasVnrArea', 'GrLivArea', 'GarageArea', 'PoolArea'], dtype='object')
```

To look specifically at the relationship between sales price and the above-ground living area of houses, you can plot it by using this syntax:

**px.scatter(df, x = "GrLivArea", y = "SalePrice")**
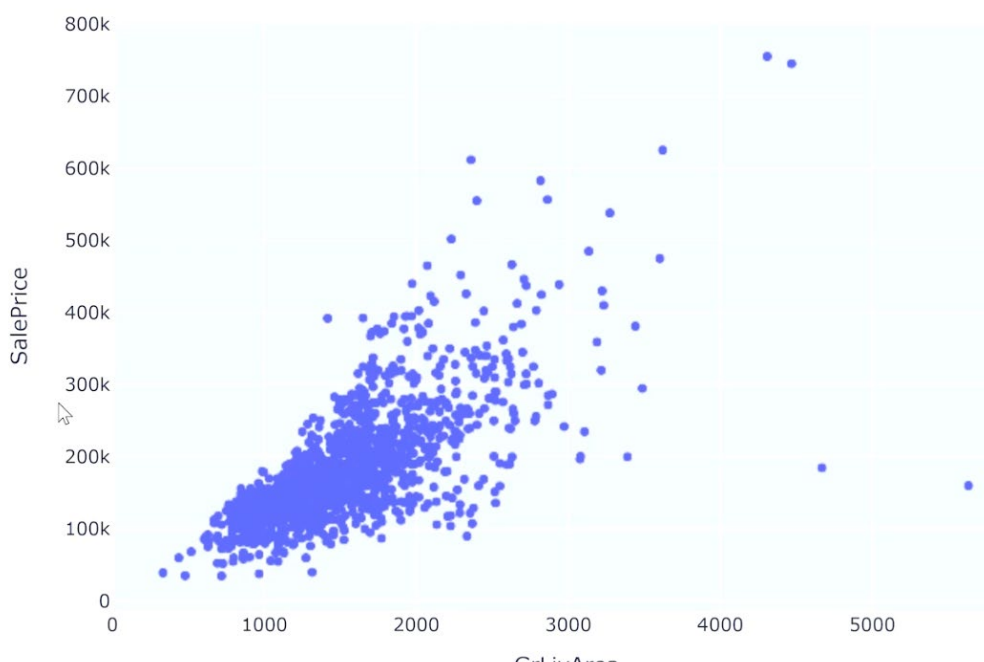
The trend in the plot confirms that houses with a bigger square footage sell for higher prices. You can continue to do this with different features to see the relationship it has to the pricing of houses.

You can also add other dimensions to a plot.

For instance, you can add the overall quality feature and the number of bathrooms to better visualize how the different features of a house work together to affect the sales price:

px.scatter(df, x = "GrLivArea", y = "SalePrice", color = "OverallQual", size = "FullBath")
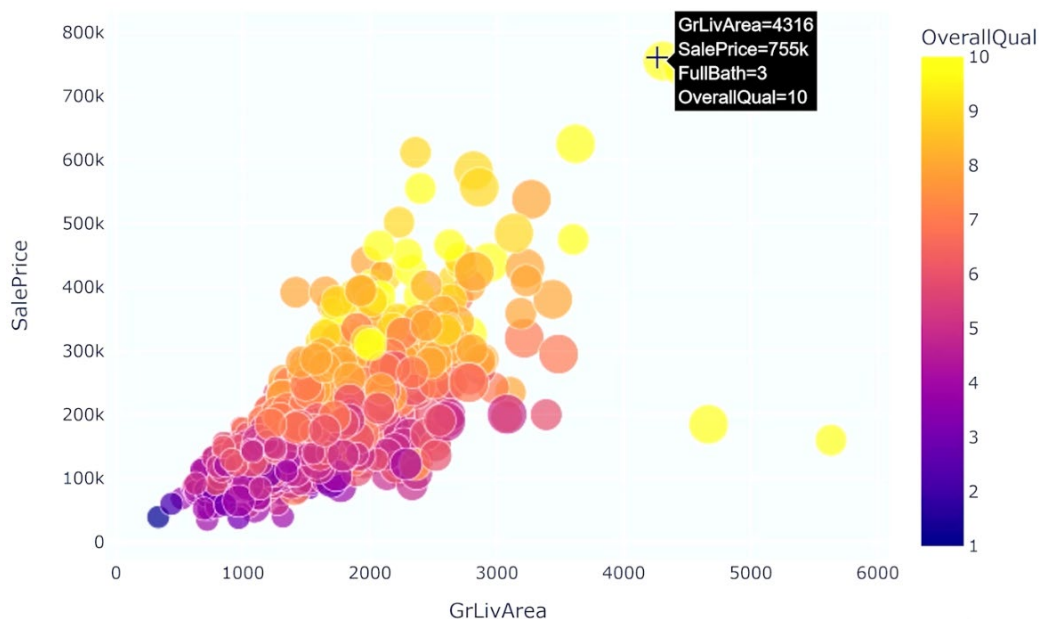
Exploring the data in order to understand it better, enables you to create machine learning models that can give accurate predictions.

The Consequences of Using the Wrong Data

Data science helps you to build models that can find hidden patterns in the data, make predictions, or help to explain the world in a new way. And the excitement around machine learning and data science is based on the potential power of computational models to create artificial intelligence.

But the over-reliance on technology and algorithms can lead you to forget the most crucial ingredient: **the right data**.

For a data science project to be successful, you need to start with the right questions, like:

- What is the right data for this problem?
- How do you find the right data?

Data science should be a means to an end, not the end in itself. Before you develop a model you need to:

- Understand the problem you are trying to solve
- Know what data you have
- Know what data you would ideally like to have

Choosing the **wrong data** can cause **big problems**, and predictive algorithms in health care is a good example to illustrate how easily these problems can arise.

There is huge excitement about the potential power of AI to predict unfavorable health outcomes. The ability to make such predictions will grant doctors an opportunity to intervene and improve people's health to save not only money for the health system, but people's lives as well.

However, a study performed on one widely-used predictive algorithm, showed a problem in its predictive model. The model was supposed to assess risk and assign interventions when a patient was above a certain threshold. To determine that threshold, you need a measure of health risk, and the metric the developers used for identifying illness, was how much money was spent on individuals. This algorithm was widely adopted and impacted care for around 200 million Americans.

However, the study found the predictive algorithm to be **racially biased**, and that bias stemmed from the choice of data in use. Since black patients use less health care — due to the history of racism in the medical industry as well as a range of socio-economic factors — focusing on cost as a measure of health left black patients looking like they were at lower risk of illness.

But when you look at a better measure of actual illness, such as the true number of chronic conditions, it seems that black patients are, in fact, at a higher risk of illness.

This example points to the importance of choosing the right data, and truly considering what your strategic goal is before you start building, testing, and using your data science models.