Data Science for Everyone

TRIO Lyon College Summer 2024

Marcus Birkenkrahe

June 13, 2024

Contents

1 Zoom

- Open a browser to zoom.us
- Click on `Join` at the top
- Enter 388 023 5098
- Launch meeting in browser
- Don't bother with audio ("Continue")
- Watch and code along

2 Sources

This file is available as an Emacs Org-mode notebook on the software development platform GitHub at: tinyurl.com/trio-data-science

The inspiration and some of the content for this session comes from Bradford Tuckfield's book "Dive into Data Science" (NoStarch, 2023).

The presented material is very similar to what you'll hear in my "Introduction to data science" class (taught every fall at Lyon).

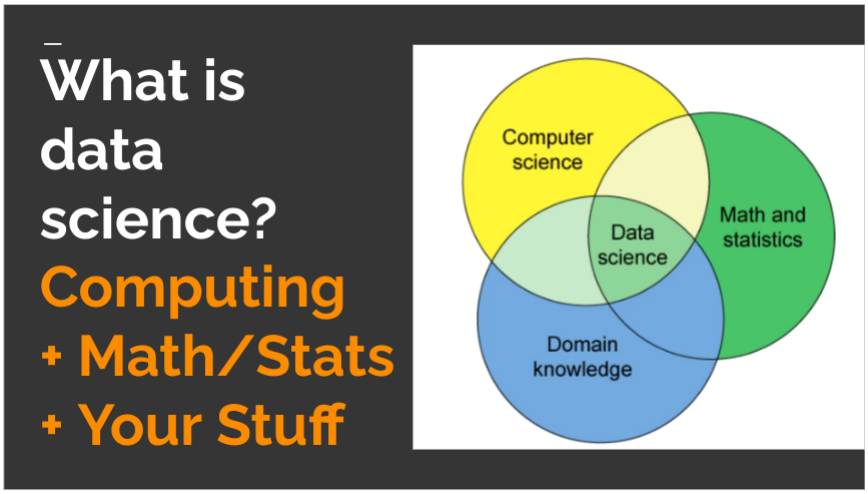
Some of the slides come from Lyon's overview presentation on the computer and data science program, see www.lyon.edu/data-science.

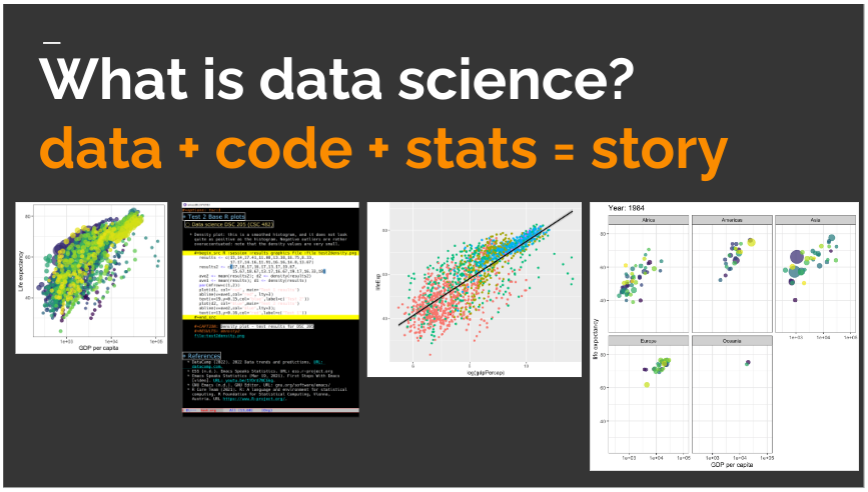
3 Overview

1. What is data science and how is it done?
2. What is exploratory data analysis (EDA)?
3. What is a business scenario and why is it important?

4. How can you find patterns in datasets?
5. How can you store, import and display data with Python?
6. How can you calculate statistics and analyse data?
7. How can you draw and display simple plots?
8. How can you customize and improve plots?
9. How can you plot subsets of data?
10. How can you test different plot types?

4 What is data science?





In terms of computer science and programming language, three languages are indispensable for data scientists: R, Python, and SQL. You'll learn all three of these (and many more) at Lyon College.

5 Why is data science so hot right now?

What do you think?

Here is what I think:

1. The recent generative AI hype is based on transformers, deep machine learning models and natural language processing, which are part of (algorithmic) data science -> machine learning
2. The availability of large datasets and the relative ease of storing and processing the data when using fast computing equipment and networks -> databases
3. The desire to automate more and more simple processes so that humans can save their energy and brains to address complex problems -> robotics
4. The commercial interest of a small number of very large, very powerful tech companies that have smelled profits and ride the top of the hype wave (even though they also don't really know what's what). -> hype cycle

6 What is literate programming?

You've already seen it: it's the combination of data + code + stats that enables you to tell a story about the data much more easily.

Today, you're going to use Google Colaboratory, an interactive notebook application, to code alongside me. This is also how I always teach!

Open colab.research.google.com now and follow my lead to explore the notebook capabilities:

1. Create a headline using markdown (`# headline`).
2. Format text as code, bold-face and italics.
3. Add code blocks and edit source code.
4. Execute code blocks and generate output on the fly.

The notebook saves you enormous amounts of work related to the often complicated infrastructure when pursuing data science projects:

- Mastering the file system (where everything is stored on the PC)
- Mastering the shell (connection to the operating system)
- Mastering the installation of libraries
- Mastering the control of graphics
- Getting coding help

You will still have to learn all these things, but not on this day!

In short, literate programming is the usual way in which data scientists explore data and build reports for humans by combining documentation, code, and output.

7 Exploring and analysing data

- *Exploratory Data Analysis* means using different tools to look at the data in as much detail as possible to understand
    1. where the data come from
    2. how large is the dataset
    3. what the data contain

4. what the data mean
5. what quality the data have
6. what format the data have
7. what information might be obtained from the data
8. what the logical next step(s) could be

- Suppose that someone gives you the following dataset:

```
model,mpg,cyl,disp,hp,drat,wt,qsec,vs,am,gear,carb
Mazda RX4,21,6,160,110,3.9,2.62,16.46,0,1,4,4
Mazda RX4 Wag,21,6,160,110,3.9,2.875,17.02,0,1,4,4
Datsun 710,22.8,4,108,93,3.85,2.32,18.61,1,1,4,1
Hornet 4 Drive,21.4,6,258,110,3.08,3.215,19.44,1,0,3,1
Hornet Sportabout,18.7,8,360,175,3.15,3.44,17.02,0,0,3,2
Valiant,18.1,6,225,105,2.76,3.46,20.22,1,0,3,1
Duster 360,14.3,8,360,245,3.21,3.57,15.84,0,0,3,4
Merc 240D,24.4,4,146.7,62,3.69,3.19,20,1,0,4,2
Merc 230,22.8,4,140.8,95,3.92,3.15,22.9,1,0,4,2
Merc 280,19.2,6,167.6,123,3.92,3.44,18.3,1,0,4,4
Merc 280C,17.8,6,167.6,123,3.92,3.44,18.9,1,0,4,4
Merc 450SE,16.4,8,275.8,180,3.07,4.07,17.4,0,0,3,3
Merc 450SL,17.3,8,275.8,180,3.07,3.73,17.6,0,0,3,3
Merc 450SLC,15.2,8,275.8,180,3.07,3.78,18,0,0,3,3
Cadillac Fleetwood,10.4,8,472,205,2.93,5.25,17.98,0,0,3,4
Lincoln Continental,10.4,8,460,215,3,5.424,17.82,0,0,3,4
Chrysler Imperial,14.7,8,440,230,3.23,5.345,17.42,0,0,3,4
Fiat 128,32.4,4,78.7,66,4.08,2.2,19.47,1,1,4,1
Honda Civic,30.4,4,75.7,52,4.93,1.615,18.52,1,1,4,2
Toyota Corolla,33.9,4,71.1,65,4.22,1.835,19.9,1,1,4,1
Toyota Corona,21.5,4,120.1,97,3.7,2.465,20.01,1,0,3,1
Dodge Challenger,15.5,8,318,150,2.76,3.52,16.87,0,0,3,2
AMC Javelin,15.2,8,304,150,3.15,3.435,17.3,0,0,3,2
Camaro Z28,13.3,8,350,245,3.73,3.84,15.41,0,0,3,4
Pontiac Firebird,19.2,8,400,175,3.08,3.845,17.05,0,0,3,2
Fiat X1-9,27.3,4,79,66,4.08,1.935,18.9,1,1,4,1
Porsche 914-2,26,4,120.3,91,4.43,2.14,16.7,0,1,5,2
Lotus Europa,30.4,4,95.1,113,3.77,1.513,16.9,1,1,5,2
Ford Pantera L,15.8,8,351,264,4.22,3.17,14.5,0,1,5,4
Ferrari Dino,19.7,6,145,175,3.62,2.77,15.5,0,1,5,6
```

```
Maserati Bora,15,8,301,335,3.54,3.57,14.6,0,1,5,8
Volvo 142E,21.4,4,121,109,4.11,2.78,18.6,1,1,4,2
```

- Can you make sense of the data at all? (Without external help.)
- What we can say about the data without further exploration:
    1. the first line looks different from all the other lines
    2. the dataset consists of 33 lines, 566 words, and 1766 characters
    3. it contains text, integer and decimal numbers
    4. The first word in the last 32 lines could be the name of a car
    5. All items in the dataset are separated by commas
- What this could mean:
    1. `mtcars` has a headline and 32 records
    2. `mtcars` data are stored as a CSV (Comma Separated Values) file
    3. `mtcars` contains data about different car models
    4. `mtcars` contains data about car `model`, miles-per-gallon (`mpg`), horsepower (`hp`), automatic/manual (`am`), number of gears (`gear`)
- Where do the data come from?

    `mtcars` is a built-in dataset from the base version of the R language. I obtained the data from GitHub. The data was supposedly "extracted from the 1974 Motor Trend US magazine, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973-1974 models)." (Source: stat.ethz.ch). You can easily find good and bad EDA examples with `mtcars`[1].
- In the case of `mtcars`, we can answer all these questions by just looking at the entire dataset. But real datasets are rarely this small, which is why we need tools to help us explore the data.

[1] Here is one where `mtcars` was used to demonstrate the capabilities of an R package called `explore` (part of a vignette for this package). The original source for the dataset is given as Henderson & Velleman (1981).

- By the way, how does "exploration" differ from "analysis"? And does it matter?

  > "Exploration" comes from the Latin word 'explorare', which means 'to call out' when searching or to seek information.
  >
  > "Analysis", Greek "" (análisis), means breaking something up or dissolving it in order to understand the whole through its parts.
  >
  > EDA then is short for seeking out information by looking at details of the data. This will give you insights but an important so-called "Gestalt" (German for 'shape') principle says that "the whole is more than the sum of its parts" - analysis is followed by synthesis where you put the parts together again after having understood them.
  >
  > It is perhaps important to note that the process of gaining insights through dissection followed by re-composition works well for machines but not so well for organic entities like humans, or their most precious properties - a brain, or a thought, do not lend themselves easily to such a simple procedure.

8 Devising a business scenario

- Though we found out where the `mtcars` dataset comes from, it's removed from us in space, time, and meaning: we're not readers of the 1970s magazine, and most of us don't drive these classic cars.

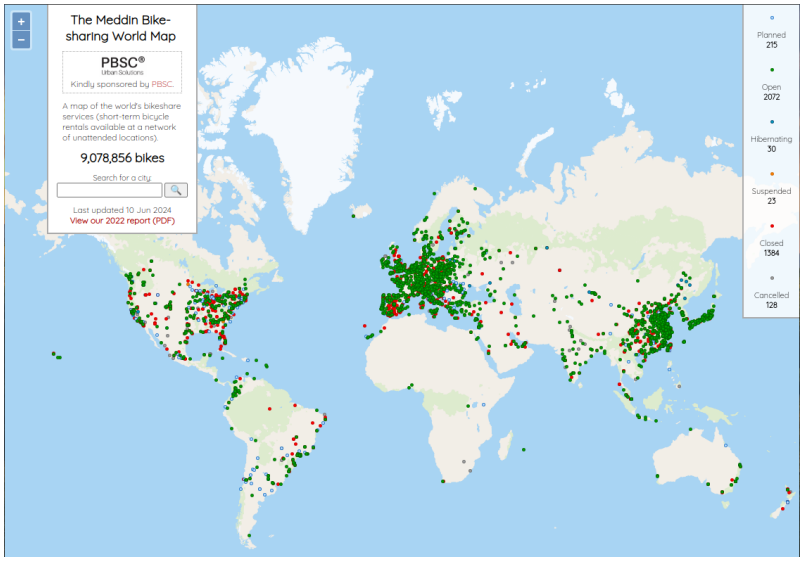
- Why does this matter?

  > Direct knowledge of the data matters because it helps you imbue the data with meaning. Data aren't like atoms, they're more like story particles that form patterns, which you have to discover and interpret.

- Instead of `mtcars`, we start with a different scenario that you may be more familiar with:

  > You've joined a company, BikeShare Inc, which rents bicycles to people to ride around the city. The goals of the company are the same as for other companies. They include customer satisfaction, employee morale, brand recognition,

market share, cost reduction, and revenue growth. You could probably find this company and its business locations on the "Bike-sharing World Map" (I bet you didn't know that such a map existed).



- For our purpose of EDA, these goals correspond to *metrics*, entities that can be measured (at least indirectly): we can't know if customers are really satisfied but we can make them fill in a survey, and we don't know 'employee morale' directly but we can measure how many employees stay with the company for how long, etc.

- Your mission is to pick an area to attend to in order to improve the performance of the company[2]. You've decided to dive into the data yourself to understand better how the company works (and where it could be run better).

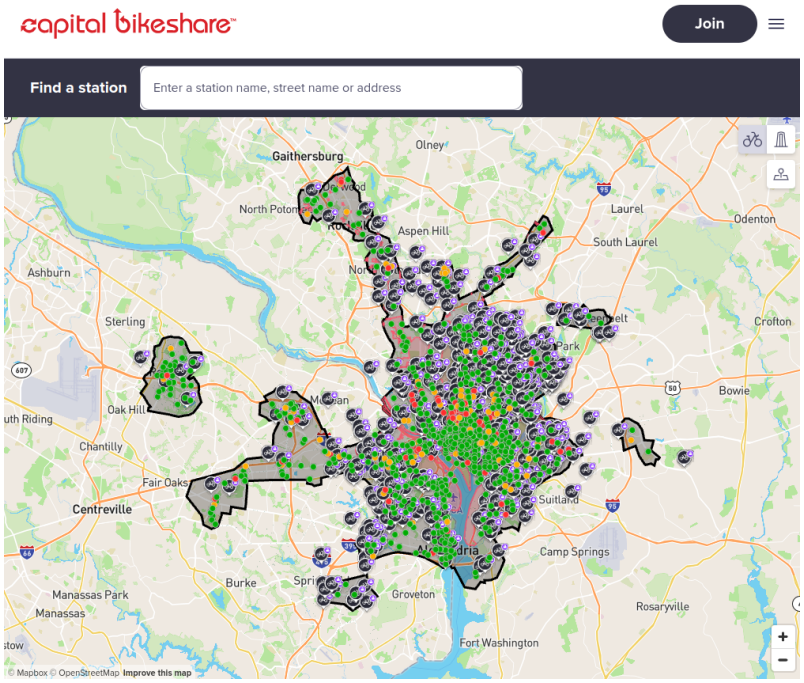
- What you've just heard is a business scenario, or a narrative, a story that gives meaning to the data. A side effect is that you care more about the data than you would if you had no idea[3]

[2]If that's an honorable or desirable goal is another question: some people, especially in the US, might say that bikes aren't very practical in the US, while in Europe for example, bicycles are very common in cities (but the cities look very differently - for one thing, they're a lot more packed and distances - e.g. to work or to the shops - are generally shorter, and the weather is more temperate).

[3]Of course, the scenario could also have the opposite effect: for example, passionate

9 Finding patterns in Datasets

- You can download real bike-sharing data from here: tinyurl.com/hour-csv[4]



- Take a look at the data (not the map): what do you see?

  Multiple metrics visible in the headline. All data seem to be either numbers (integer and decimal) or dates. There are 17,379 records. The dataset is much too large to be analyzed by the naked eye alone. The format of the dataset is CSV, Comma-Separated-Values: the values are separated by commas.

peace activists may not wish to work with data of arms manufacturers; people of one culture may not care about data about people from another culture etc. Therefore, it helps if, in addition to an interest in the data, you also enjoy getting to know, and master, computational and mathematical tools of data science.

[4] I obtained the data from Tuckfield (2023). He notes that the original source is Capital Bikeshare for Washington D.C., and that the data was compiled and augmented (?) and posted by others at a non-disclosed location.

- You can look at the data with a spreadsheet application, e.g. Google Sheets: tinyurl.com/hours-sheets. Now, the columns are easily distinguishable but the data is not easier to manage.

- **What does the dataset represent?**

    1. Each row of the dataset represents information about a particular hour between 12 am on Jan-1-2011 and 11:59 pm on Dec-31-2012 - more than 17,000 hours.
    2. Each column of the dataset shows a particular metric measured for each of these hours, e.g. wind speed measured at a particular weather station
    3. The data have been transformed: for example, wind speed has been transformed from miles-per-hour to a number in (0,1) so that 0 corresponds to no wind, and 1 to fast wind speed.
    4. The last three columns are the most important ones for the company - the number of people who used the bikes each hour:

`casual`	people who used bikes without registering
`registered`	people who register for discounts and benefits
`count`	total number of people who used bikes

- Look at the data again (only the first 24 hours) to see if you can discern any patterns in these customer related columns:

dteday	season	yr	mnth	hr	holiday	weekday	workingday	weathersit	temp	atemp	hum	windspeed	casual	registered	count
2011-01-01	1	0	1	0	0	6	0	1	0.24	0.2879	0.81	0	3	13	16
2011-01-01	1	0	1	1	0	6	0	1	0.22	0.2727	0.8	0	8	32	40
2011-01-01	1	0	1	2	0	6	0	1	0.22	0.2727	0.8	0	5	27	32
2011-01-01	1	0	1	3	0	6	0	1	0.24	0.2879	0.75	0	3	10	13
2011-01-01	1	0	1	4	0	6	0	1	0.24	0.2879	0.75	0	0	1	1
2011-01-01	1	0	1	5	0	6	0	2	0.24	0.2576	0.75	0.0896	0	1	1
2011-01-01	1	0	1	6	0	6	0	1	0.22	0.2727	0.8	0	2	0	2
2011-01-01	1	0	1	7	0	6	0	1	0.2	0.2576	0.86	0	1	2	3
2011-01-01	1	0	1	8	0	6	0	1	0.24	0.2879	0.75	0	1	7	8
2011-01-01	1	0	1	9	0	6	0	1	0.32	0.3485	0.76	0	8	6	14
2011-01-01	1	0	1	10	0	6	0	1	0.38	0.3939	0.76	0.2537	12	24	36
2011-01-01	1	0	1	11	0	6	0	1	0.36	0.3333	0.81	0.2836	26	30	56
2011-01-01	1	0	1	12	0	6	0	1	0.42	0.4242	0.77	0.2836	29	55	84
2011-01-01	1	0	1	13	0	6	0	2	0.46	0.4545	0.72	0.2985	47	47	94
2011-01-01	1	0	1	14	0	6	0	2	0.46	0.4545	0.72	0.2836	35	71	106
2011-01-01	1	0	1	15	0	6	0	2	0.44	0.4394	0.77	0.2985	40	70	110
2011-01-01	1	0	1	16	0	6	0	2	0.42	0.4242	0.82	0.2985	41	52	93
2011-01-01	1	0	1	17	0	6	0	2	0.44	0.4394	0.82	0.2836	15	52	67
2011-01-01	1	0	1	18	0	6	0	3	0.42	0.4242	0.88	0.2537	9	26	35
2011-01-01	1	0	1	19	0	6	0	3	0.42	0.4242	0.88	0.2537	6	31	37
2011-01-01	1	0	1	20	0	6	0	2	0.4	0.4091	0.87	0.2537	11	25	36
2011-01-01	1	0	1	21	0	6	0	2	0.4	0.4091	0.87	0.194	3	31	34
2011-01-01	1	0	1	22	0	6	0	2	0.4	0.4091	0.94	0.2239	11	17	28
2011-01-01	1	0	1	23	0	6	0	2	0.46	0.4545	0.88	0.2985	15	24	39

    1. The number of `registered` users is mostly greater than the number of `casual` users.

2. The two groups peak at slightly different times (1 pm vs. 2 pm).

- What can you do with insights like these?

    1. This could mean that using the service casually isn't as easy as it could be to increase casual users.
    2. This could indicate demographic differences between the groups (e.g. age), suggesting different marketing approaches.

- Just by looking at a few columns of the first day of the data, we have already learnt a few things about the company and are starting to get some business ideas. No math, (almost) no tools, just common sense so far!

10 Using CSV to store data

- The data that you see in the CSV file tinyurl.com/hour-csv are called "raw" data though they're minimally formatted already, because every data item is a character of text.

- What does the spreadsheet add to this?

    1. Alignment in columns for readability
    2. Flexibility in moving the columns around for better viewing
    3. Computations on numerical data
    4. Opening depends on available (commercial) spreadsheet software

- The advantage of CSV:

    1. Files can be easily created
    2. Files can be easily opened by many different programs
    3. Data can easily be changed
    4. Files are small, portable, easy to share

11 Displaying data with Python

To display the data in a different way and open it up for analysis, we import them into Python.

- **Python** is not the only but an obvious choice: it's a FOSS language that can easily be learnt and that is widespread (in fact the most popular among all high level programming languages)[5].
- **What does "import into Python" mean?** It means that we **read** the CSV data into a format that will allow us to use Python's advanced functionality to explore further and analyse more thoroughly.
- **Advanced functionality** means that Python has *functions*, pre-written sets of instructions that can compute new quantities for us: for example an average over many values, or create a graph showing us the evolution of a quantity (like count) over time.
- These are the steps required to display the data using Python:
  1. **install** a Python library
  2. **import** the Python library
  3. **read** the CSV file into Python's format
  4. **store** the Python-formatted file in a Python object
  5. **print** the Python object to the screen
- **Why so many steps?** Each of these has a reason:
  1. **Install**: Extra functionality are not contained in base Python: if you want to do special things, you need special tools.
  2. **Import**: Python is interactive - you need to make the library, which you now have on your computer, available in the current Python session.
  3. **Read**: The imported Python library has a special function that can understand CSV and spit it out in the format Python needs.
  4. **Print**: Python has a built-in function to display data.

[5] Another popular choice would be R, which is also FOSS, even easier to learn, and better equipped for statistics and visualization. A third possibility is to stick with the CSV format and use text mining tools on the command line, so-called *shell* commands, which are super-fast, small and simple.

- In summary, the multi-step process is owed to the fact that there are many interlocking parts to achieve something that seems simple to the unsuspecting user. The prize is having a lot of power over how to display and analyse the data.
- Here's the code to achieve this[6]:

```
import pandas as pd
hour = pd.read_csv('data/hour.csv')
print(hour.head())
```

- Let's dissect the code: it is important, now and forever, that you understand every detail of your code down, every character, its position and meaning[7].
- It is useful (especially at the start) to add the explanation to the code in the code block in the form of comments that Python ignores:

```
################################################
# Python script to display the top of a CSV file #
################################################

# Import the pandas library and alias it as pd
import pandas as pd

# Read CSV file from its location and store data in a DataFrame
hour = pd.read_csv('data/hour.csv')

# Print the top of the DataFrame
print(hour.head())
```

- I introduced a few additional things:

[6] I've left out the *install* step because this is done outside of Python using the `pip` package manager program. To install the `pandas` package, run `pip install pandas` on your computer. Of course, you first have to have (and possibly install) the `pip` program.

[7] One way of achieving this is to read new code (and especially code written by others) *from the bottom* and *from the right*, i.e. the wrong way around: in this way, your brain has to make sense of what you read rather than delude itself into understanding what it doesn't understand. One way of stopping yourself from learning how to code is to always copy and paste blindly, or (worse) let generative AI do it for you. This works only for the simplest of programs. Once you are an expert, you may make good use of AI.

1. `DataFrame` is the format of the Python `pandas` library for tabular data
2. When reading data, the computer needs to be given an exact location. On my computer, the CSV file is in the directory `data`, so I need to specify `data/hours.csv` for it to be found.
3. Aliasing the `pandas` library as `pd` means that anything that's in the library must be addressed using `pd`. `read_csv` is a *method* (or function) inside the library. For the computer to find it, I must write `pd.read_csv`. If I only wrote `read_csv`, I'd get an error.
4. I assigned the data to a `pandas` `DataFrame` named `hours`: If I want to use `pandas` functions on `hours`, I need to tell the computer that, too: I must write `hours.head` to let it know that I want to run `head` on `hours`. If I only wrote `head`, I'd get an error.

- You should try it for yourself and see what happens if you violate these rules. You can see that even the simplest of operations requires an enormous amount of background knowledge. You cannot really do without it but you also don't have to learn it all on one day.
- We could have saved ourselves the use of `head` and simply written `print(hour)` - but that would have given us a display of the whole dataset, which is huge (you should try this, too).
- How does the output compare to the CSV file and the spreadsheet?
  1. Data is arranged by column similar to the spreadsheet
  2. Some columns in the middle are left out and replaced by ellipses
  3. Only the first five rows are displayed plus the headline

12 Calculating summary statistics

- Summary statistics are mathematical functions that reveal properties, which make the more sense the more data we have.
- Such properties include the average, the median, the maximum, the minimum, and the standard deviation. Here, we will not go into the math (it's not difficult) but only present code and results.
- Begin by calculating the mean of one of the columns, `count`:

```
print(hour['count'].mean())
```

```
189.46308763450142
```

- The command follows the same rule as `hour.head()` earlier: this time, we apply the function `mean`, which computes the average - but it makes no sense to average the whole table: `hour['count']` selects only the `count` column out of the table and average over its values.

- Since we don't need this level of precision (this many numbers after the decimal point), let's store the average in a variable `mean` and print it with 2 decimals after the decimal point:

```
mean = hour['count'].mean()
print(f'{mean:.2f}')
```

```
189.46
```

- So in 2011-2012, 189.46 bikes were taken out per hour. This gives us an idea of the size of the business. According to their website, a single ride with Capital bikeshare cost $0.05 per minute (or $3 per hour): the average hourly revenue is therefore 189.46 x $3 = $568.388, or almost $5 mio per year. This is not the business profit, of course since running the business, buying and maintaining bikes, paying insurance etc. is not free but it's still a feasible business[8].

- Let's compute a few more measures: median (or middle magnitude) and standard deviation (a measure of spread) for `count`, and minimum and maximum for `registered`:

```
# compute statistical measures
median = hour['count'].median()
std_dev = hour['count'].std()
min_reg = hour['registered'].min()
max_reg = hour['registered'].max()

# print results
```

[8] In case you were wondering if bike sharing is a business at all, according to Mordor Intelligence, the global 2024 bike sharing market has a size of $7.85 billion, projected to increase to $12.44 billion over the next 5 years. Such a prediction is, however, dependent on many volatile factors, e.g. the need for urban transportation, the reduction in costs, and state subsidization.

```
print(f'Median count:{median:.2f}')
print(f'Standard deviation of count:{std_dev:.2f}')
print(f'Minimum number of registered users:{min_reg}')
print(f'Maximum count:{max_reg}')
```

```
Median count:142.00
Standard deviation of count:181.39
Minimum number of registered users:0
Maximum count:886
```

- The average is quite far away from the median, which suggests that the spread of the data is high, and that there may be outliers. In general, the median is a better measure for centrality in this case.
- The summary shows that there are hours when no registered users are present, and the difference between the maximum and the mean and median also shows that the data is quite spread out.
- The standard deviation shows the spread most clearly: the smaller this number, the closer together are the data.
- You can also get summary statistics more quickly with `pandas =describe` method, which lists summaries for all numeric columns:

```
print(hour.describe())
```

```
  instant        season  ...    registered         count
count  17379.0000  17379.000000  ...  17379.000000  17379.000000
mean    8690.0000      2.501640  ...    153.786869    189.463088
std     5017.0295      1.106918  ...    151.357286    181.387599
min        1.0000      1.000000  ...      0.000000      1.000000
25%     4345.5000      2.000000  ...     34.000000     40.000000
50%     8690.0000      3.000000  ...    115.000000    142.000000
75%    13034.5000      3.000000  ...    220.000000    281.000000
max    17379.0000      4.000000  ...    886.000000    977.000000

[8 rows x 16 columns]
```

- Here, `count` is the total number of records or rows of data used for the computations. `25%`, and `75%` are the first and the third quartile, and `50%` is the median: for example, 25% of the hours in the dataset had 40 users or fewer, while 75% had more.

- Some of these make no sense for the variables: `season` for example is a *categorical* variable, a finite set {1,2,3,4}. For such variables, none of the statistical summaries are meaningful.

- Summary stats can be used to quickly verify data validity: for example, if an experiment with people reports an average age of 200 for the participants, something is wrong. Such errors are quite common in research.

- Besides checking the data, summary stats are important for business decisions: e.g. you could reduce prices during the night to reduce the number of hours with lower ridership, or you could reward operators in whose shift the maximum ridership is surpassed.

- Much of what follows goes more deeply into the data, and in parallel, into the business. To do this, we must isolate subsets of data and look which patterns we can find in them.

- Data science happens between these two poles: the dataset as a whole, which must be managed, imported, stored, etc., and subsets of the data, which correspond with parts of the world. Any story worth telling has large and small aspects, just like a human has a character, and also individual traits worth looking at.

13 Analysing nighttime data

- To pursue the idea of changing pricing during the night, we need to check summary stats related to just the nighttime.

- As you might have guessed, there is not only a method for selecting columns but also a method for filtering rows from the data table, `loc`. For example, to filter the `count` data for row number `3`:

  ```
  Print(hour.loc[3,'count'])
  ```

- We had better check with the dataset if this number is correct. Can you recall, how we printed the first 5 rows of the dataset `hour`?

  ```
  print(hour.head())
  ```

```
   instant      dteday  season  yr  ...  windspeed  casual  registered  count
0        1  2011-01-01       1   0  ...        0.0       3          13     16
1        2  2011-01-01       1   0  ...        0.0       8          32     40
2        3  2011-01-01       1   0  ...        0.0       5          27     32
3        4  2011-01-01       1   0  ...        0.0       3          10     13
4        5  2011-01-01       1   0  ...        0.0       0           1      1

[5 rows x 17 columns]
```

- You find the value 13 in the `count` column in the row indexed by 3 (which is the fourth column because we start counting at 0 [9]).
- A table always has rows and columns: the square brackets `[ ]` are an **index operator** with two arguments, `[rows,columns]`. So `[3,'count']` extracts the table elements with a row index of `3` and the column name `count`.
- Since `count` also happens to be the column number `16`, the following command would give the same result. Notice that it uses `iloc` and not `loc`:

```
print(hour.iloc[3, 16])
```

```
13
```

- How can you know that `count` is column number 16 except by counting manually using for example the `columns` command?

```
print(hour.columns)
```

```
Index(['instant', 'dteday', 'season', 'yr', 'mnth', 'hr', 'holiday', 'weekday',
       'workingday', 'weathersit', 'temp', 'atemp', 'hum', 'windspeed',
       'casual', 'registered', 'count'],
      dtype='object')
```

- We need a test that goes through the column labels and checks which one is `count` - the `get_loc` method from the `pandas` Index object does the trick:

[9] This is normal for most programming languages, including C, C++, Java and JavaScript, and it has historical reasons. The language R starts counting at 1 instead, which I always found very helpful.

```
print(hour.columns.get_loc('count'))
```

```
16
```

- Or you could write a little function yourself that goes through the `Index` object (converted to a `list`) returned by `columns`, and checks each label:

```
Index = list(hour.columns)
index = 0
for i in Index:
    if i=='count': print(index)
    index = index + 1
```

```
16
```

- Wrap this in a function:

```
# function definition
def getloc(dataframe, label):
    '''Return positional index for dataframe label
    dataframe: a DataFrame
    label: string label for dataframe column
    '''
    Index = list(dataframe.columns)
    index = 0
    for i in Index:
        if i==label: return index
        index = index + 1

# function call
print(getloc(hour,'count'))
print(getloc(hour,'registered'))
```

```
16
15
```

- We can also check a **range of values** by using the **colon** operator (`:`) - for example, to extract rows [indexed] 2 to 4 from the `registered` column, we would write:

```
print(hour.loc[2:4,'registered'])
print(hour.iloc[2:5,15])
```

```
2    27
3    10
4     1
Name: registered, dtype: int64
2    27
3    10
4     1
Name: registered, dtype: int64
```

- Notice another difference between `loc` and `iloc`: the latter leaves out the the last index after the colon! For a complete comparison, see the `pandas` online documentation.
- The process of filtering a subset of data is called *subsetting*.
- Within the `loc` method, you can use *logical conditions*, that is you can filter values based on a logical check with a logical operator.
- Example: in the following code chunk we filter all rows from the `registered` column whose `hr` value is smaller than 5, or 12pm to 4am, and then we average over them:

```
print(f"{hour.loc[hour['hr'] < 5, 'registered'].mean():.2f}")
```

```
20.79
```

- The answer: on average, 20 to 21 bikes were taken out in the small hours of the morning.
- With multiple conditions we can achieve more detail: what if we want to check, which nighttime rentals took place while the temperature was relatively cold or relatively warm? Logically, we're looking for two conditions to both hold, requiring the AND (`&`) operator.
- Instead of packing statements, it's good practice to store intermediate results before printing them:

```
cold = hour.loc[ (hour['hr'] < 5) & (hour['temp'] < .50), 'count']
warm = hour.loc[ (hour['hr'] < 5) & (hour['temp'] > .50), 'count']
print(f"Average users at night when it was cold: {cold.mean():.2f}")
print(f"Average users at night when it was warm: {warm.mean():.2f}")
```

```
Average users at night when it was cold: 19.52
Average users at night when it was warm: 33.64
```

- We can also check inclusive conditions, for example to find out how many bikes were taken out on average when the temperature was warm OR (`|`) the humidity was high:

```
print(f"{hour.loc[(hour['temp']>0.5)|(hour['hum']>0.5),'count'].mean():.2f}")
```

```
193.37
```

14 Analysing seasonal data

- Another business improvement strategy could target a `season`, which is recorded in the data as 1 for winter, 2 for spring, 3 for summer and 4 for fall.
- The `groupby` method for `pandas` groups all records according to a value or set of values:

```
print(hour.groupby(['season'])['count'].mean())
```

```
season
1    111.114569
2    208.344069
3    236.016237
4    198.868856
Name: count, dtype: float64
```

- Let's dissect the command whose individual parts should be clear by now:
  1. `hour.groupby` calls the `groupby` method on the `hour` DataFrame
  2. `hour.groupby(['season'])` creates groups for the `season` values

3. `hour.groupby(['season']['count']` selects the `count` column
4. `hour.groupby(['season']['count'].mean()` averages over each group of the subset.

- The result shows the average ridership in winter (1), spring (2), summer (3), and fall (4), and a definite pattern: higher ridership in spring and summer, and lower ridership in winter and fall.

- We can also group multiple columns: first by `season` and then by `holiday`: additional column labels are listed in the `groupby` argument:

```
print(hour.groupby(['season','holiday'])['count'].mean())
```

```
season  holiday
1       0          112.685875
1           72.042683
2       0          208.428472
1          204.552083
3       0          235.976818
1          237.822917
4       0          199.965998
1          167.722222
Name: count, dtype: float64
```

- Here, the hourly data are first split by `season` and then the result for each `season` is split into holidays (1) and non-holidays (0). We notice that holidays don't make a positive difference in fall and winter.

- In a similar way, you can take other columns, analyze them by asking questions, and link the results to business decisions.

15 Drawing and displaying a simple plot

- Displaying data in a tabular format with columns, headlines etc. is already a form of visualization. Another approach is making plots.

- It is fairly easy in most languages (except SQL and bash who are too specialized on their focus of databases and system commands, resp.) to create simple graphics, which is all what we're after here.

- For more specialized, highly customized, or animated graphics, there are separate packages available, which often require substantial time investment. At Lyon, there is an extra course on "data visualization" that teaches this stuff.

- Here are the minimal steps to make a plot

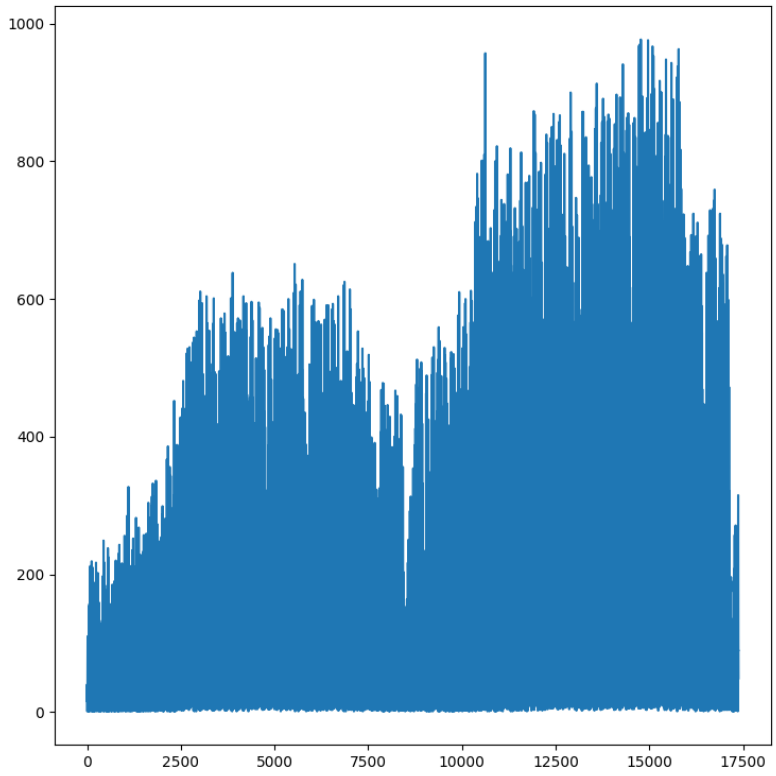
  1. Decide what you want to plot
  2. Import a Python library that knows how to plot to our current session
  3. Decide what type of plot to make
  4. Select the data for the plot
  5. Create the plot
  6. Display the plot

- Example: Let's say we want to see how the `count` values, the total number of rides, varied over time: this means that `count` is our dependent, and `instant` (which is a running label for the hours) is our independent variable. Every data point is a pair, and therefore a *scatterplot* (points scattered across the canvas) is suitable to show this pattern [10].

- Here's the code with comments:

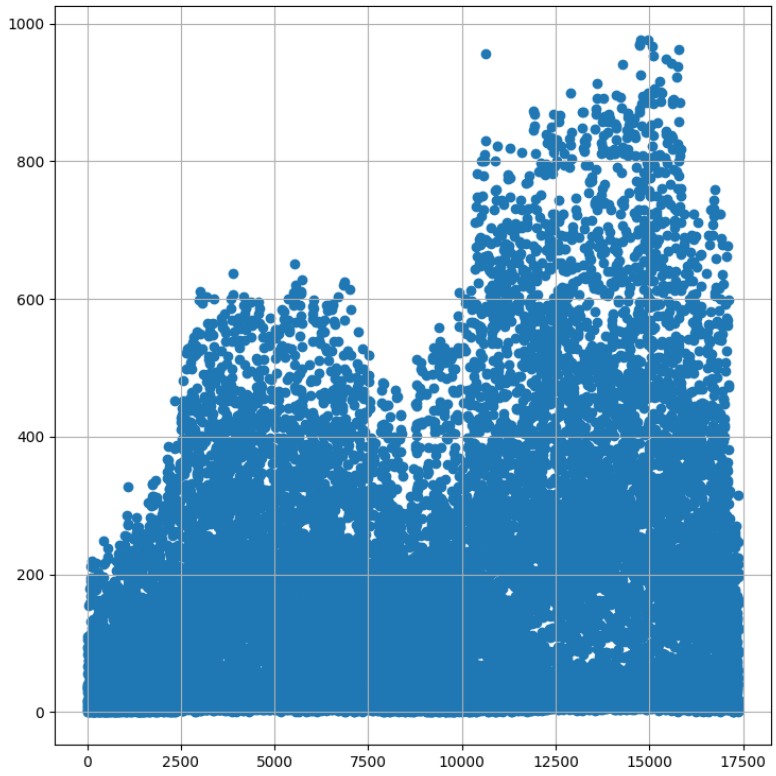
```
# import graphics library
import matplotlib.pyplot as plt
# clear graphics
plt.clf()
# define variables
x = hour['instant'] # independent variable
y = hour['count'] # dependent variable
# create the plot
plt.plot(x,y)
# display the plot
plt.tight_layout()
plt.savefig("dids1.png") # or plt.show() outside of Org-mode
```

[10] Following the change of a variable like `count` over time is also called a *time series* - that's an important data structure for example in finance or climate research where important quantities vary over time. Every interesting quantity varies over time, of course, but if we explicitly show the change of one quantity against time, that's a time series.



- This plot looks more like a painting than a scattering of points: this is because there are so many data points - all the rides taken out (y-axis) plotted for every hour of two years (x-axis), and because the default plot is a lineplot. Using `scatter` instead of `plot` gives us an impression of scattered points.

```
plt.clf()
plt.scatter(x,y)
plt.grid()
plt.tight_layout()
plt.savefig("dids2.png")
```



- What information can you get from this plot?

    1. **Seasonal variation**: a year has about 8760 hours - the middle point of the graph shows a clear minimum. This type of graph with two distinct hills is also called *bimodal*.
    2. **Overall trend**: the corresponding seasons show that the second year of operation was a great deal more successful than the first, almost by 50%.
    3. **Statistical summaries**: You cannot read their values off readily but looking at the gridlinds, you can confirm several of the summary statistics by order of magnitude.

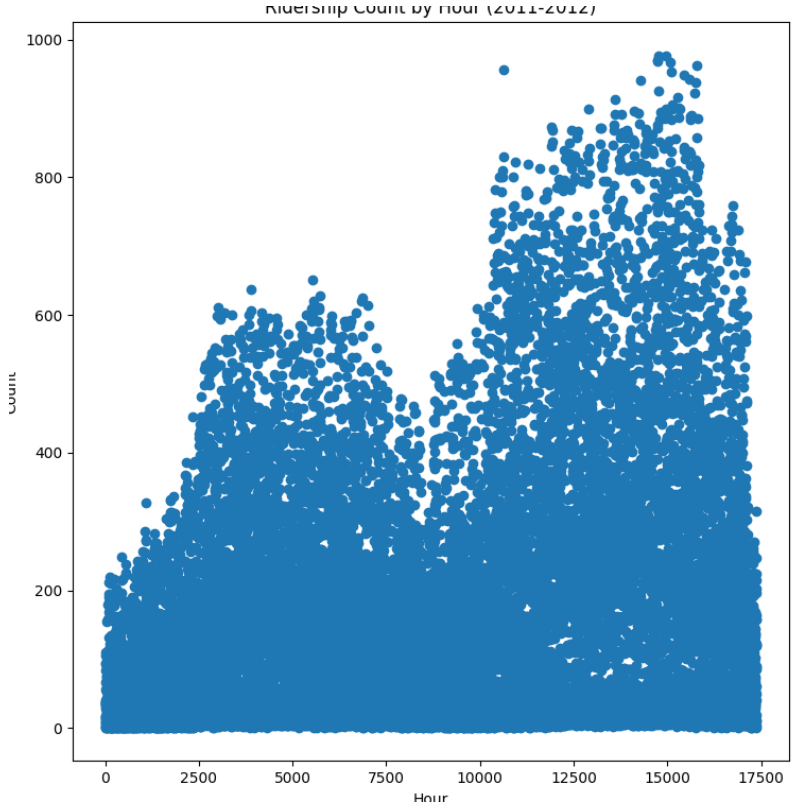
```
print(365*24)
```

```
8760
```

16 Clarifying plots with titles and labels

- It wouldn't be easy to explain this plot to someone else. To clarify the presentation, we can add labels and a title to the plot.
- We only need to add a few extra methods. Notice that we're cleaning the canvas and redrawing the figure:

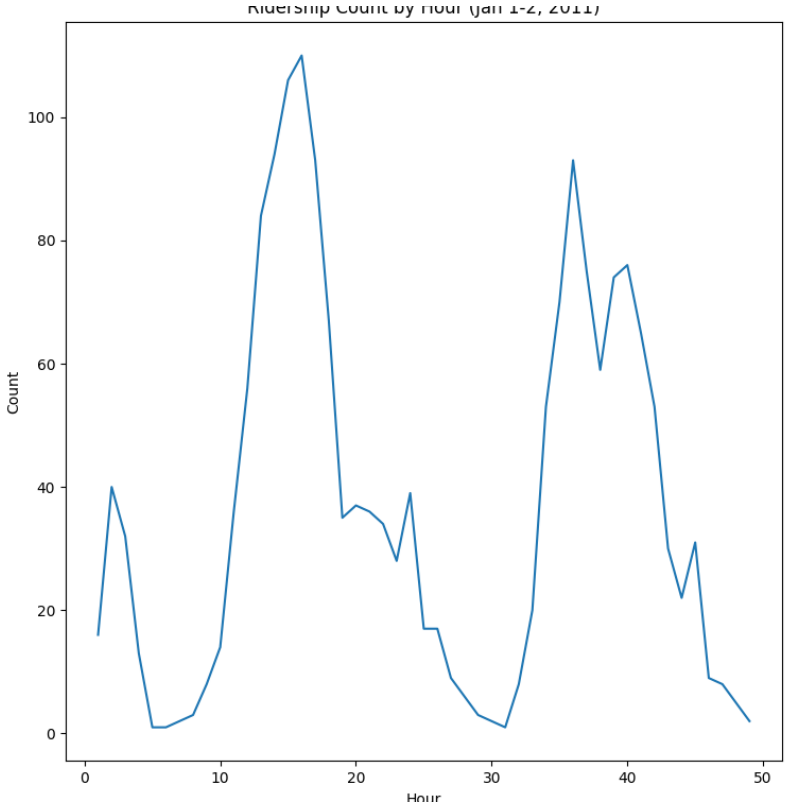
```
plt.clf()
plt.scatter(x,y)
plt.xlabel("Hour")
plt.ylabel("Count")
plt.title("Ridership Count by Hour (2011-2012)")
plt.savefig("dids3.png")
```



17 Plotting subsets of data

- The dataset is very large, and looking at all the data at once is hard. We use subsetting to plot a smaller subset, for example the first 48 hours of 2011 - the second argument to `loc` (`:`) includes *all* columns:

```
# subset dataset
hour_first_48 = hour.loc[0:48,:]
# data for plotting
x = hour_first_48['instant']
y = hour_first_48['count']
# plotting
plt.clf()
plt.plot(u,v)
plt.xlabel("Hour")
plt.ylabel("Count")
plt.title("Ridership Count by Hour (Jan 1-2, 2011)")
plt.savefig("dids4.png")
```

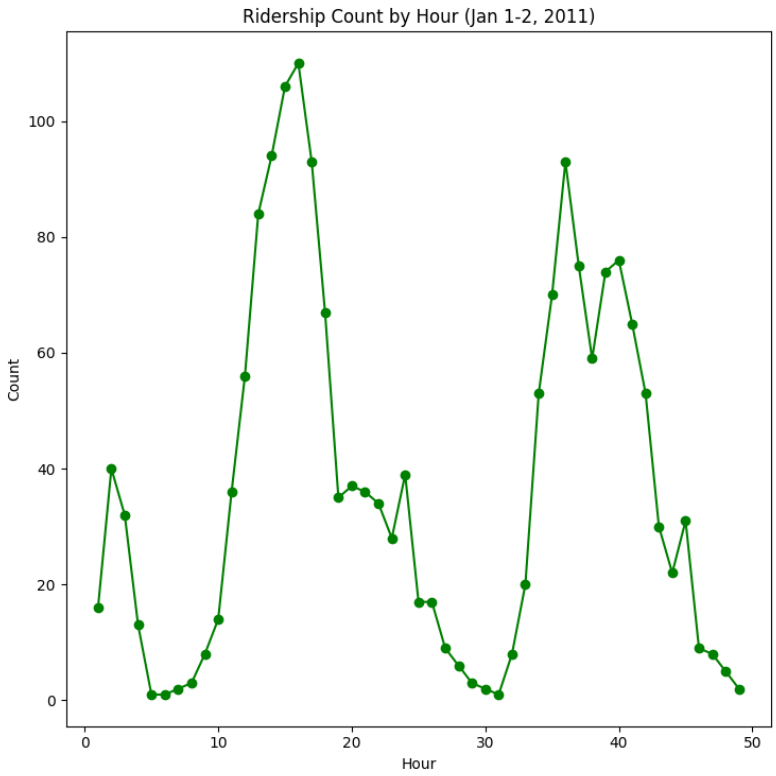


18 Testing different plot types

- To properly understand and train yourself in the use of functions like `plot`, you must read the online documentation.
- There are many ways to alter the appearance of a plot. For example, you can a `marker` parameter, which has multiple values to change the data points, and the `color` parameter to change the color of the graph. (Source).
- In the next plot, we draw the data points as filled circles and change the color to green:

```
plt.clf()
plt.plot(x,y, color='green', marker='o')
plt.xlabel("Hour")
plt.ylabel("Count")
```

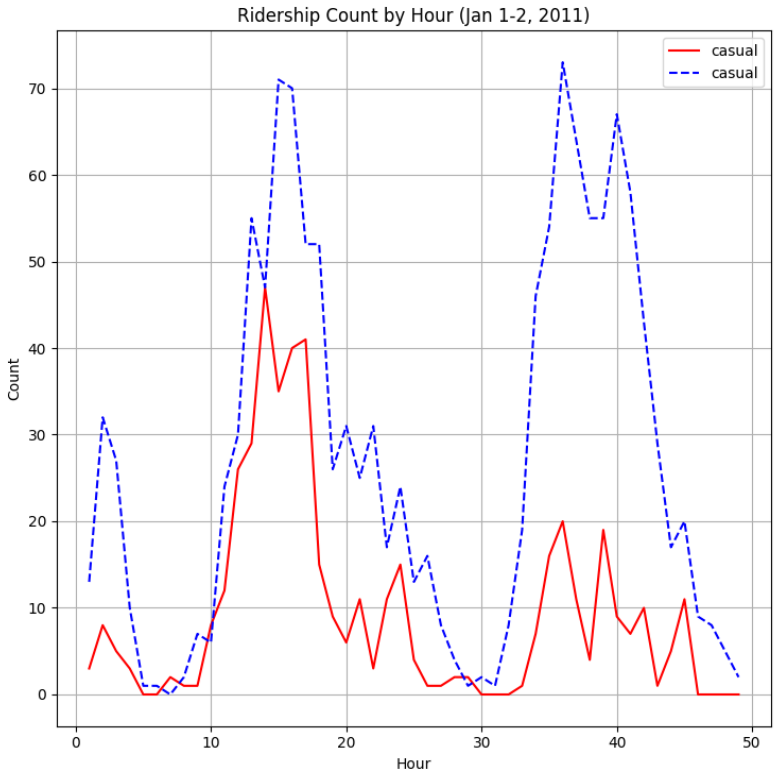
```
plt.title("Ridership Count by Hour (Jan 1-2, 2011)")
plt.tight_layout()
plt.savefig("dids5.png")
```



- You can alter the line type as well. In the next plot, we show `casual` and `registered` riders over the first 2 days of the dataset, distinguished by color and linetype. We also introduce the `legend` method, which automatically adds a legend for every `label` and `linestyle`.

```
# subsetting
y1 = hour_first_48['casual']
y2 = hour_first_48['registered']
# clear plotting canvas
plt.clf()
## first plot: casual riders
plt.plot(x,y1, color='red', label='casual',linestyle='-')
```

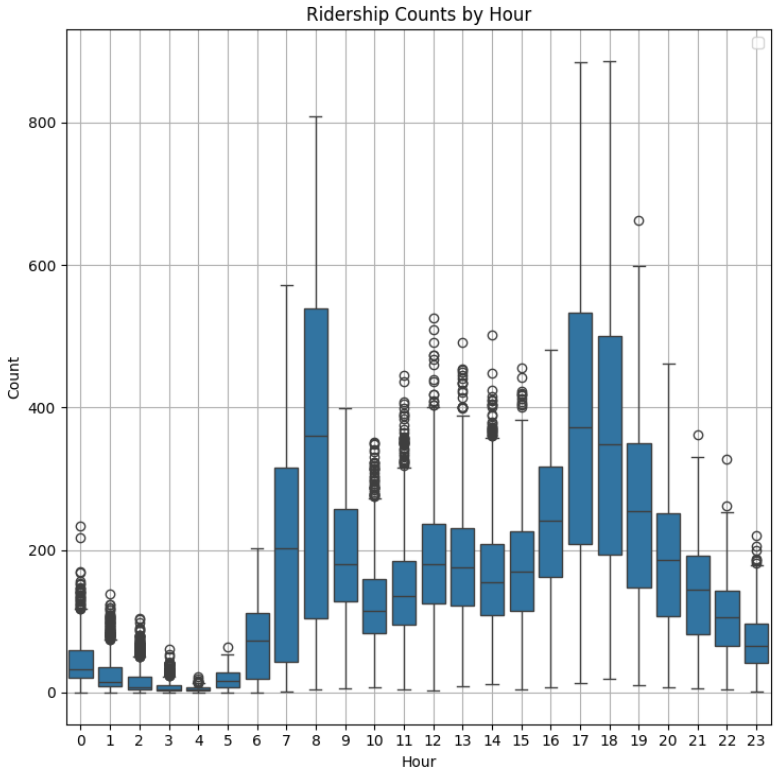
```
## second plot: registered riders
plt.plot(x,y2, color='blue', label='casual',linestyle='--')
## labels and title
plt.xlabel("Hour")
plt.ylabel("Count")
plt.title("Ridership Count by Hour (Jan 1-2, 2011)")
plt.legend()
plt.grid()
plt.tight_layout()
plt.savefig("dids6.png")
```



- The plot shows that the number of casual riders is almost always lower than the number of registered riders. This plot would benefit from a legend that explains what the graphs mean.
- So far we've only seen scatterplots and lineplots. Another interesting plot type is the *box plot*, also called "box and whiskers" plot.

- To draw the boxplot, we'll load another package, `seaborn`, which works alongside `matplotlib` (documentation):

```
# import seaborn package
import seaborn as sns
# plotting
plt.clf()
sns.boxplot(x='hr',
            y='registered',
            data=hour)
# labelling
plt.xlabel("Hour")
plt.ylabel("Count")
plt.title("Ridership Counts by Hour")
plt.legend()
plt.grid()
plt.tight_layout()
plt.savefig("dids7.png")
```



- Let's analyze the plot:

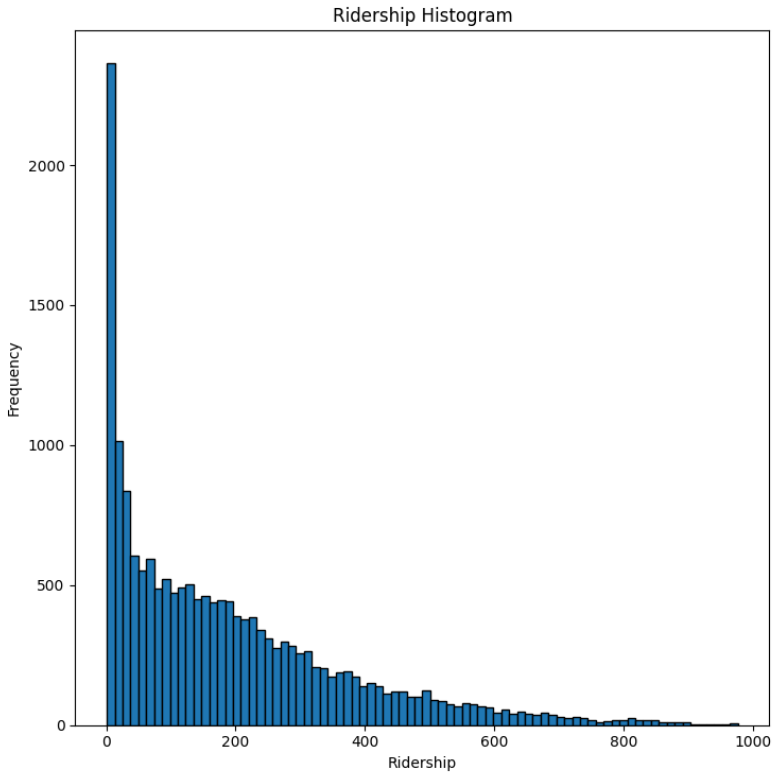
  1. the independent x variable is the hour column of our `hour` `DataFrame`
  2. for each hour, the statistical summary is computed and shown as a box with whiskers: the lower whisker is the minimum, the upper whisker is the maximum, the lower and upper edge of the box are the 25% and 75% percentile respectively, and the bar across the box is the median (or 50% percentile).
  3. The individual data points shown above several of the boxes are *outliers*, values that don't "fit in the box" and that are too far away from the average.
  4. The format of the `seaborn` `boxplot` is slightly different: you can pass the name of the `DataFrame` to the function using the `data` parameter.

- You can now compare ridership at different times of day. You can see

the impact of rush hour or work commutes between 5-7 am and 5-6 pm.

- When you're interested in frequency or number of counts of a numeric variable, a *histogram* is a useful plot. It uses the `bins` parameter to specify the number of bins where each bin contains a range of values (documentation).

```
plt.clf()
plt.hist(hour['count'], bins=80, edgecolor='black')
plt.xlabel("Ridership")
plt.ylabel("Frequency")
plt.title("Ridership Histogram")
plt.tight_layout()
plt.savefig("dids8.png")
```



- Since the number of bins is fixed, the total ridership is divided equally among the total number of bins. You can see that there are way more

riders in the first bin in our data: for more than 2,000 hours, ridership was very low. For 500 hours, ridership was around 100.

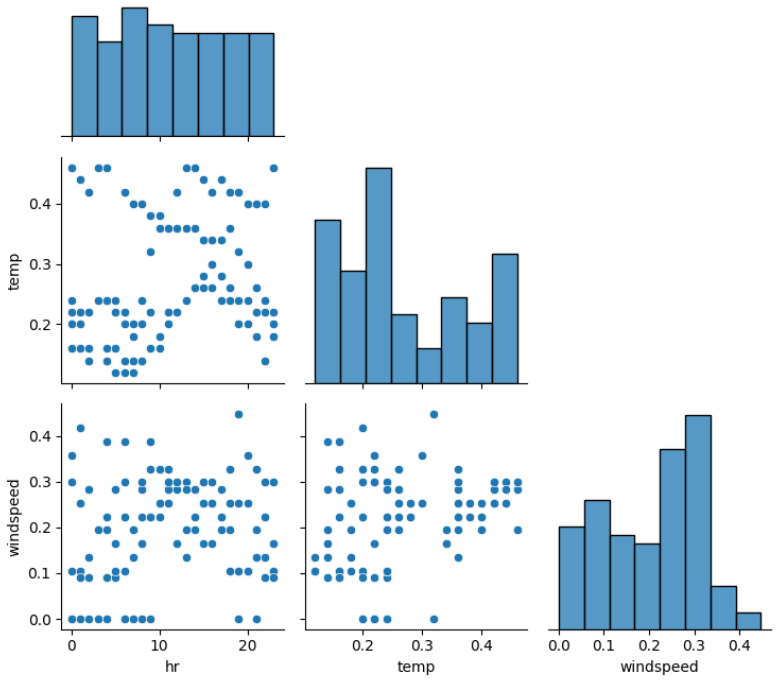
- A histogram could be used to think about the capacity of the company: if you have 1,000 bicycles available, you could probably sell 200 of them since very few hours have more than 800 bikes rented out.

- A *pair plot* pairs more than one pair of variables: it draws every possible scatterplot for every possible pair of variables in your data. For this plot, you don't need to specify the labels:

```
# variables to pair up
vars = ['hr', 'temp', 'windspeed']

# subsetting
hour_first_100 = hour.loc[0:100, vars]

# plotting
plt.clf()
sns.pairplot(hour_first_100, corner=True)
plt.tight_layout()
plt.savefig("dids9.png")
```



- The plot contains both scatterplots and histograms. There aren't clear patterns among the data points in the scatterplots - temperature, windspeed and hour don't seem to be strongly correlated, that is they don't exhibit a strong tendency to grow or fall together.
- The study of variables that are coupled is an important part of data science because it allows us to indirectly draw conclusions from one feature of the data to another feature (for example: holidays and ridership); in many cases, we can let go of variables that are so strongly correlated that they don't provide new information (that's good because carrying variables along eats up computing power); and it allows us to simplify visualizations and storytelling.
- Highly correlated variables are like groups of people where you only care about one person so having to deal with the group may be boring - even though you may at other times care about the group.

19 Summary

- Exploratory Data Analysis (EDA) involves using various tools to deeply understand the data's origin, size, content, meaning, quality, format, potential insights, and logical next steps.
- Establishing a business scenario provides context and meaning to the data, making it more relevant and engaging for analysis.
- Analyzing a dataset, such as bike-sharing data, helps identify patterns and insights that can inform business decisions and strategies.
- CSV files are simple, portable, and widely supported for storing raw data, although spreadsheet applications offer enhanced readability and computational flexibility.
- Importing and displaying data in Python involves reading CSV files into a format suitable for analysis, using libraries like pandas.
- Summary statistics, such as mean, median, and standard deviation, provide quick insights into data properties and potential business implications.
- Subsetting data allows focused analysis on specific conditions, such as nighttime ridership, to explore patterns and inform decisions.
- Grouping data by categories, like season and holiday, reveals trends and variations that can guide business strategies.
- Creating plots in Python, such as scatterplots and line plots, visualizes data trends and patterns, aiding in analysis and communication.
- Adding labels and titles to plots enhances clarity and helps convey the data's story effectively.
- Subsetting large datasets for specific time frames or conditions provides more manageable and focused visualizations.
- Experimenting with various plot types, like box plots, histograms, and pair plots, helps uncover different aspects of the data and enhances analysis.

20 Glossary

Term/Command	Definition
EDA	Exploratory Data Analysis
CSV	Comma-Separated Values; format for tabular data in plain text.
pandas	A Python library for data manipulation and analysis.
DataFrame	A 2-dimensional labeled data structure in pandas.
.read$_{\text{csv}}$()	pandas method to read a CSV file into a DataFrame.
.head()	pandas method to return the first n rows of a DataFrame.
.mean()	pandas method to calculate the mean of a DataFrame column.
.median()	pandas method to calculate the median of a DataFrame column.
.std()	pandas method to calculate the standard deviation.
.min()	pandas method to return the minimum value of a column.
.max()	pandas method to return the maximum value of a column.
.describe()	pandas method to generate descriptive statistics.
.loc[]	pandas method to access rows and columns by labels/booleans.
.iloc[]	pandas method to access rows and columns by integer positions.
.groupby()	pandas method to group DataFrame.
matplotlib	Python library for creating visualizations.
plt.clf()	matplotlib function to clear the current figure.
plt.plot()	matplotlib function to plot data on a 2D graph.
plt.scatter()	matplotlib function to create a scatter plot.
plt.xlabel()	matplotlib function to set the label for the x-axis.
plt.ylabel()	matplotlib function to set the label for the y-axis.
plt.title()	matplotlib function to set the title of the plot.
plt.legend()	matplotlib function to display a legend on the plot.
plt.grid()	matplotlib function to display grid lines on the plot.
plt.savefig()	matplotlib function to save the current figure to a file.
seaborn	Python visualization library based on matplotlib.
sns.boxplot()	seaborn function to create a box plot.
sns.pairplot()	seaborn function to create a pair plot.
.groupby()	pandas method to group DataFrame.
plt.hist()	matplotlib function to create a histogram.
edgecolor	Parameter in plt.hist() to set the color of the bin edges.
.get$_{\text{loc}}$()	pandas Index method to get integer location for label.