Contents

Chapter 1

Data Tables and pandas

Much of the data that you will encounter in your career will come to you as a table. Some of these tables are spreadsheets, some are in relational databases, some will come to you as CSV files.

Typically each column will represent an attribute (like height or acreage) and each row will represent an entity (like a person or a farm). You might get a table like this:

property_id	bedrooms	square_meters	estimated_value
7927	3	921.4	$ 294,393
9329	2	829.1	$ 207,420

Typically, one of the columns is guaranteed to be unique. We call this the *primary key*. In this table, `property_id` is the primary key: every property has one, and no two properties have the same `property_id`.

1.1 Data types

Each column in a table has a type, and these usually correspond pretty nicely with types in Python.

Here are some common datatypes:

Type	Python type	Example
Integer	int	`910393`
Float	float	`-23.19`
String	string	`'Fred'`
Boolean	bool	`False`
Date	datetime.date	`2019-12-04`
Timestamps	datetime.datetime	`2022-06-10T14:05:22Z`

Sometimes it is OK to have values missing. For example, if you had a table of data about employees, maybe one of the columns would be `retirement`, a date that tells you when the person retired. People who had not yet retired would have no value in this column. We would say that they have *null* for `retirement`.

Sometimes there are constraints on what values can appear in the column. For example, if the column were `height`, it would make no sense to have a negative value.

Sometimes a column can only be one of a few values. For example, if you ran a bike rental shop, each bicycle's status would be "available", "rented", or "broken". Any other values in that column would not be allowed. We often call these columns *categorical*.

1.2 pandas

The Python community works with tables of data *a lot*, so it created the pandas library for reading, writing, and manipulating tables of data.

When working with tables, you sometimes need to go through them row-by-row. However, for large datasets, this is very slow. pandas makes it easy (and very fast) to say things like "Delete every row that doesn't have a value for height" instead of requiring you to step through the whole table.

In pandas, there are two datatypes that you use a lot:

- a `Series` is a single column of data.
- a `DataFrame` is a table of data: it has a `Series` for each column.

In the digital resources, you will fined `bikes.csv`. If you look at it in a text editor, it will start like this:

```
bike_id,brand,size,purchase_price,purchase_date,status
5636248,GT,57,277.99,1986-09-07,available
4156134,Giant,56,201.52,2005-01-09,rented
7971254,Cannondale,54,292.25,1978-02-28,available
3600023,Canyon,57,197.62,2007-02-15,broken
```

The first line is a header and tells you the name of each column. Then the values are separated by commas. (Thus the name: CSV stands for “Comma Separated Values”.)

1.3 Reading a CSV with pandas

Let’s make a program that reads `bikes.csv` into a pandas dataframe. Create a file called `report.py` in the same folder as `bikes.csv`.

First, we will read in the csv file. pandas has one Series that acts as the primary key; it calls this one the index. When reading in the file, we will tell it to use the `bike_id` as the index series.

If you ask a dataframe for its shape, it returns a tuple containing the number of rows and the number of columns. To confirm that we have actually read the data in, let’s print those numbers. Add these lines to `report.py`:

```
import pandas as pd

# Read the CSV and create a dataframe
df = pd.read_csv('bikes.csv', index_col="bike_id")

# Show the shape of the dataframe
(row_count, col_count) = df.shape
print(f"*** Basics ***")
print(f"Bikes: {row_count:,}")
print(f"Columns: {col_count}")
```

Build it and run it. You should see something like this:

```
*** Basics ***
Bikes: 998
Columns: 5
```

Note that your table actually had 6 columns. The index series is not included in the shape.

1.4 Looking at a Series

Let's get the lowest, the highest, and the mean purchase price of the bikes. The purchase price is a series, and you can ask the dataframe for it. Add these lines to the end of your program:

```
# Purchase price stats
print("\n*** Purchase Price ***")
series = df["purchase_price"]
print(f"Lowest:{series.min()}")
print(f"Highest:{series.max()}")
print(f"Mean:{series.mean():.2f}")
```

Now when you run it, you will see a few additional lines:

```
*** Purchase Price ***
Lowest:107.37
Highest:377.7
Mean:249.01
```

What are all the brands of the bikes? Add a few more lines to your program that shows how many of each brand:

```
# Brand stats
print("\n*** Brands ***")
series = df["brand"]
series_counts = series.value_counts()
print(f"{series_counts}")
```

Now when you run it, your report will include the number of bikes for each brand from most common to least:

```
*** Brands ***
Canyon        192
BMC           173
Cannondale    170
Trek          166
GT            150
Giant         147
Name: brand, dtype: int64
```

`value_counts` returns a Series. To format this better we need to learn about accessing individual rows in a series.

1.5 Rows and the index

In an array, you ask for data using an the location (as an int) of the item you want. You can do this in pandas using `iloc`. Add this to the end of your program:

```
# First bike
print("\n*** First Bike ***")
row = df.iloc[0]
print(f"{row}")
```

When you run it, you will see the attributes of the first row of data:

```
*** First Bike ***
brand                      GT
size                       57
purchase_price         277.99
purchase_date      1986-09-07
status              available
Name: 5636248, dtype: object
```

Notice that the data coming back is actually another series.

The last line says that the name (the value for the index column) for this row is 5636248. In pandas, we usually use this to locate particular rows. For example, there is a row with `bike_id` equal to 2969341. Let's ask for one entry from the

```
print("\n*** Some Bike ***")
brand = df.loc[2969341]['brand']
print(f"brand = {brand}")
```

Now you will see the information about that bike:

```
*** Some Bike ***
brand = Cannondale
```

pandas has a few different ways of getting to that value. All of these get you the same thing:

```
brand = df.loc[2969341]['brand'] # Get row, then get value
brand = df['brand'][2969341]     # Get column, then get value
brand = df.loc[2969341, 'brand'] # One call with both row and value
```

1.6 Changing data

One of your attributes needs cleaning up. Every bike should have a status and it should be one of the following strings:"available", "rented", or "broken". Get counts for each unique value in status:

```
print("\n*** Status ***")
series = df["status"]
missing = series.isnull()
print(f"{missing.sum()} bikes have no status.")
series_counts = series.value_counts()
for value in series_counts.index:
    print(f"{series_counts.loc[value]} bikes are \"{value}\"")
```

This will show you:

```
*** Status ***
7 bikes have no status.
389 bikes are "rented"
304 bikes are "broken"
296 bikes are "available"
1 bikes are "Flat tire"
1 bikes are "Available"
```

Right away we can see two easily fixable problems: Someone typed "Available" instead of "available". Right after you read the CSV in, fix this in the data frame:

```
mask = df['status'] == 'Available'
print(f"{mask}")
df.loc[mask, 'status'] = 'available'
```

When you run this, you will see that the mask is a series with `bike_id` as the index and `False` or `True` as the value, depending on whether the row's status was equal to "Available".

When you use `loc` with this sort of mask, you are saying "Give me all the rows for which the mask is True." So, the assignment only happens in the one problematic row.

Let's get rid of the mask variable and do the same for turning `Flat tire` into `Broken`:

```
df.loc[df['status'] == 'Available', 'status'] = 'available'
df.loc[df['status'] == 'Flat tire', 'status'] = 'broken'
```

Now those problems are gone:

```
7 bikes have no status.
389 bikes are "rented"
305 bikes are "broken"
297 bikes are "available"
```

What about the rows with no values for status? We were pretty certain that the bikes were available, we could just set them to ’available’:

```
missing_mask = df['status'].isnull()
df.loc[missing_mask, 'status'] = 'available'
```

Or maybe we would print out the IDs of the bikes so that we could go look for them:

```
missing_mask = df['status'].isnull()
missing_ids = list(df[missing_mask].index)
print(f"These bikes have no status:{missing_ids}")
```

But lets just keep the rows where the status is not null:

```
missing_mask = df['status'].isnull()
df = df[~missing_mask]
```

At the end of your program, write out the improved CSV:

```
df.to_csv('bikes2.csv')
```

Run the program and open `bikes2.csv` in a text editor.

1.7 Derived columns

Let’s say that you want to add a column with age of the bicycle in days:

```
bike_id,brand,size,purchase_price,purchase_date,status,age_in_days
5636248,GT,57,277.99,1986-09-07,available,13061
4156134,Giant,56,201.52,2005-01-09,rented,6362
7971254,Cannondale,54,292.25,1978-02-28,available,16174
```

Your first problem is that the purchase_date column looks like a date, but really it is a string. So you need to convert it to a date. You can do this by applying a function to every item in the series:

```
df['purchase_date'] = df['purchase_date'].apply(lambda s: datetime.date.fromisoformat(s))
```

(With pandas, there is often more than one way to do things. pandas has a to_datetime function that converts every entry in a sequence to a datetime object. So here is another way to convert the string column in to a date column:

```
df['purchase_date'] = pd.to_datetime(df['purchase_date']).dt.date
```

You can look up dt and date if you are curious.)

Now, we can use the same trick to create a new column with the age in days:

```
today = datetime.date.today()
df['age_in_days'] = df['purchase_date'].apply(lambda d: (today - d).days)
```

When you run this, the new bikes.csv will have an age_by_date column.

Chapter 2

Data tables in SQL

Most organizations keep their data as tables inside a relational database management system. Developers talk to those systems using a language called SQL ("Structured Query Language").

Some relational database managers are pricey products you may have heard of before: Oracle, Microsoft SQL Server. Some are free: PostgreSQL or MySQL. These are server software that client programs talk to over the companies network.

There is a library, called `sqlite`, that lets us create files that hold tables. We can use SQL to create, edit, and browse those tables. sqlite is free, fast, and very easy to install. So we will use sqlite instead of a networked database management system.

If you look in your digital resources, you will find a file called `bikes.db`. I created this file using sqlite, and now you will use sqlite to access it.

In the terminal, get to the directory where `bikes.db` lives. To open the sqlite tool on that file:

```
> textbfsqlite3 bikes.db
```

(If your system complains that there is no sqlite3 tool, you need to install sqlite. See this website: https://sqlite.org/)

Please follow along: type each command shown here into the terminal and see what happens.

We mostly run SQL commands in this tool, but there are a few non-SQL commands that all start with a period. To see the tables and their columns, you can run .schema:

```
sqlite> .schema
CREATE TABLE bike (bike_id int PRIMARY KEY, brand text, size int,
                   purchase_price real, purchase_date date, status text);
```

That is the SQL command that I used to create the bike table. You can see all the columns and their types.

You want to see all the rows of data in that table?

```
sqlite> select * from bike;
4997391|GT|57|269.61|2009-05-03|rented
5429447|Cannondale|50|215.91|2002-02-17|broken
5019171|Trek|58|251.17|1985-07-11|rented
3000288|Cannondale|57|211.08|1993-01-05|broken
880965|GT|52|281.75|1995-08-02|available
...
```

You will see 1000 rows of data!

The SQL language is not case-sensitive, so you can also write it like this:

```
sqlite> SELECT * FROM BIKE;
```

Often you will see SQL with just the SQL keywords in all caps:

```
sqlite> SELECT * FROM bike;
```

The semicolon is not part of SQL, but it tells sqlite that you are done writing a command and that it should be executed.

SQL lets you choose which columns you would like to see:

```
sqlite> SELECT bike_id, brand FROM bike;
4997391|GT
5429447|Cannondale
5019171|Trek
3000288|Cannondale
...
```

Using WHERE, SQL lets you choose which rows you would like to see:

```
sqlite> SELECT * FROM bike WHERE purchase_date > '2009-01-01' AND brand = 'GT';
4997391|GT|57|269.61|2009-05-03|rented
326774|GT|56|165.0|2009-06-27|available
264933|GT|52|302.43|2009-07-09|available
5931243|GT|55|173.56|2009-11-26|rented
4819848|GT|51|221.71|2009-12-11|rented
9347713|GT|52|232.32|2009-06-13|available
3019205|GT|58|262.94|2009-08-22|available
```

Using DISTINCT, SQL lets you get just one copy of each value:

```
sqlite> SELECT DISTINCT status FROM bike;
rented
broken
available

Busted
Flat tire
good
out
Rented
```

You can also edit these rows. For example, if you wanted every status that is `Busted` to be changed to `broken`. You can use an UPDATE statement:

```
sqlite> UPDATE bike SET status='broken' WHERE status='Busted';
sqlite> SELECT DISTINCT status FROM bike;
rented
broken
available

Flat tire
good
out
Rented
```

You can insert new rows:

```
sqlite> INSERT INTO bike (bike_id, brand, size, purchase_price, purchase_date, status)
   ...> VALUES (1, 'GT', 53, 123.45, '2020-11-13', 'available');
sqlite> SELECT * FROM bike WHERE bike_id = 1;
1|GT|53|123.45|2020-11-13|available
```

You can delete rows:

```
sqlite> DELETE FROM bike WHERE bike_id = 1;
sqlite> SELECT * FROM bike WHERE bike_id = 1;
```

To get out of sqlite, type `.exit`.

Exercise 1 SQL Query

Execute an SQL query that returns the `bike_id` (no other columns) of every Trek bike that cost more than $300.

Working Space

Answer on Page 17

2.1 Using SQL from Python

The people behind sqlite created a library for Python that lets you execute SQL and fetch the results from inside a python program.

Let's create a simple program that fetches and displays the bike ID and purchase date of every Trek bike that cost more than $300.

Create a file called `report.py`:

```
import sqlite3 as db

con = db.connect('bikes.db')
cur = con.cursor()
```

```
cur.execute("SELECT bike_id, purchase_date FROM bike WHERE purchase_price > 330 AND bran
rows = cur.fetchall()

today = datetime.date.today()
for row in rows:
    print(f"Bike {row[0]}, purchased {row[1]}")

con.close()
```

When you execute it, you should see:

```
> python3 report.py
Bike 4128046, purchased 2007-08-06
Bike 7117808, purchased 1995-03-12
Bike 7176903, purchased 1986-07-03
Bike 827899, purchased 2009-03-14
Bike 363983, purchased 1970-08-16
```

Appendix A

Answers to Exercises

Answer to Exercise 1 (on page 14)

```
SELECT bike_id FROM bike WHERE purchase_price > 330 AND brand='Trek'
```

Index