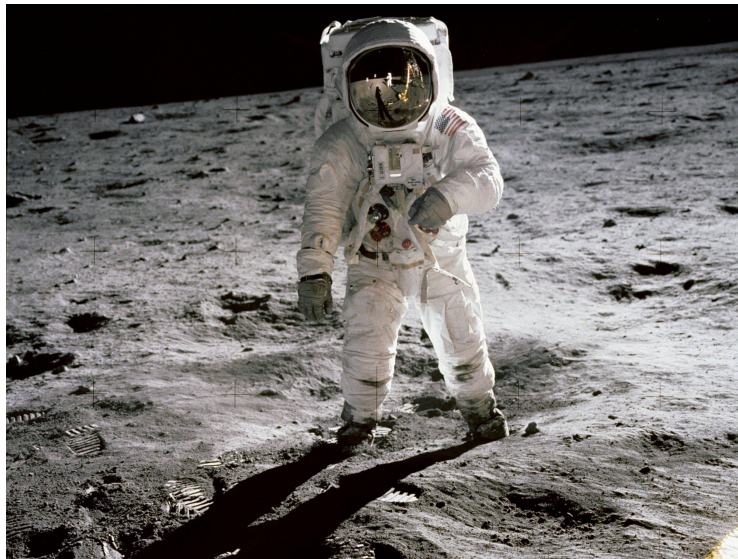


# Labs for Foundations of Applied Mathematics

Data Science Essentials

Jeffrey Humpherys & Tyler J. Jarvis, managing editors





# List of Contributors

E. Evans  
*Brigham Young University*  
R. Evans  
*Brigham Young University*  
J. Grout  
*Drake University*  
J. Humpherys  
*Brigham Young University*  
T. Jarvis  
*Brigham Young University*  
J. Whitehead  
*Brigham Young University*  
J. Adams  
*Brigham Young University*  
J. Bejarano  
*Brigham Young University*  
Z. Boyd  
*Brigham Young University*  
M. Brown  
*Brigham Young University*  
A. Carr  
*Brigham Young University*  
T. Christensen  
*Brigham Young University*  
M. Cook  
*Brigham Young University*  
R. Dorff  
*Brigham Young University*  
B. Ehlert  
*Brigham Young University*  
M. Fabiano  
*Brigham Young University*  
A. Frandsen  
*Brigham Young University*

K. Finlinson  
*Brigham Young University*  
J. Fisher  
*Brigham Young University*  
R. Fuhriman  
*Brigham Young University*  
S. Giddens  
*Brigham Young University*  
C. Gigena  
*Brigham Young University*  
M. Graham  
*Brigham Young University*  
F. Glines  
*Brigham Young University*  
C. Glover  
*Brigham Young University*  
M. Goodwin  
*Brigham Young University*  
R. Grout  
*Brigham Young University*  
D. Grundvig  
*Brigham Young University*  
J. Hendricks  
*Brigham Young University*  
A. Henriksen  
*Brigham Young University*  
I. Henriksen  
*Brigham Young University*  
C. Hettinger  
*Brigham Young University*  
S. Horst  
*Brigham Young University*  
K. Jacobson  
*Brigham Young University*

J. Leete  
*Brigham Young University*

J. Lytle  
*Brigham Young University*

R. McMurray  
*Brigham Young University*

S. McQuarrie  
*Brigham Young University*

D. Miller  
*Brigham Young University*

J. Morrise  
*Brigham Young University*

M. Morrise  
*Brigham Young University*

A. Morrow  
*Brigham Young University*

R. Murray  
*Brigham Young University*

J. Nelson  
*Brigham Young University*

E. Parkinson  
*Brigham Young University*

M. Probst  
*Brigham Young University*

M. Proudfoot  
*Brigham Young University*

D. Reber  
*Brigham Young University*

C. Robertson  
*Brigham Young University*

M. Russell  
*Brigham Young University*

R. Sandberg  
*Brigham Young University*

C. Sawyer  
*Brigham Young University*

M. Stauffer  
*Brigham Young University*

J. Stewart  
*Brigham Young University*

S. Suggs  
*Brigham Young University*

A. Tate  
*Brigham Young University*

T. Thompson  
*Brigham Young University*

M. Victors  
*Brigham Young University*

J. Webb  
*Brigham Young University*

R. Webb  
*Brigham Young University*

J. West  
*Brigham Young University*

A. Zaitzeff  
*Brigham Young University*

# Preface

This lab manual is designed to accompany the textbook *Foundations of Applied Mathematics* by Humpherys and Jarvis. While the Volume 3 text focuses on statistics and rigorous data analysis, these labs aim to introduce experienced Python programmers to common tools for obtaining, cleaning, organizing, and presenting data.

©This work is licensed under the Creative Commons Attribution 3.0 United States License. You may copy, distribute, and display this copyrighted work only if you give credit to Dr. J. Humpherys. All derivative works must include an attribution to Dr. J. Humpherys as the owner of this work as well as the web address to

<https://github.com/Foundations-of-Applied-Mathematics/Labs>

as the original source of this work.

To view a copy of the Creative Commons Attribution 3.0 License, visit

<http://creativecommons.org/licenses/by/3.0/us/>

or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.





# Contents

<b>Preface</b>	<b>iii</b>
<b>I Labs</b>	<b>1</b>
1 SQL 1: Introduction	3
2 SQL 2 (The Sequel)	15
3 Regular Expressions	23
4 Web Technologies	35
5 Web Scraping	47
6 Web Crawling	57
7 Pandas 1: Introduction	65
8 Pandas 2: Plotting	81
9 Pandas 3: Grouping	95
10 Pandas 4: Time Series	107
<b>II Appendices</b>	<b>119</b>
A Getting Started	121
B Installing and Managing Python	129
C NumPy Visual Guide	133





Part I

Labs



# 1

## SQL 1: Introduction

**Lab Objective:** *Being able to store and manipulate large data sets quickly is a fundamental part of data science. The SQL language is the classic database management system for working with tabular data. In this lab we introduce the basics of SQL, including creating, reading, updating, and deleting SQL tables, all via Python's standard SQL interaction modules.*

### Relational Databases

A *relational database* is a collection of tables called *relations*. A single row in a table, called a *tuple*, corresponds to an individual instance of data. The columns, called *attributes* or *features*, are data values of a particular category. The collection of column headings is called the *schema* of the table, which describes the kind of information stored in each entry of the tuples.

For example, suppose a database contains demographic information for  $M$  individuals. If a table had the schema (Name, Gender, Age), then each row of the table would be a 3-tuple corresponding to a single individual, such as (Jane Doe, F, 20) or (Samuel Clemens, M, 74.4). The table would therefore be  $M \times 3$  in shape. Another table with the schema (Name, Income) would be  $M \times 2$  if it included all  $M$  individuals.

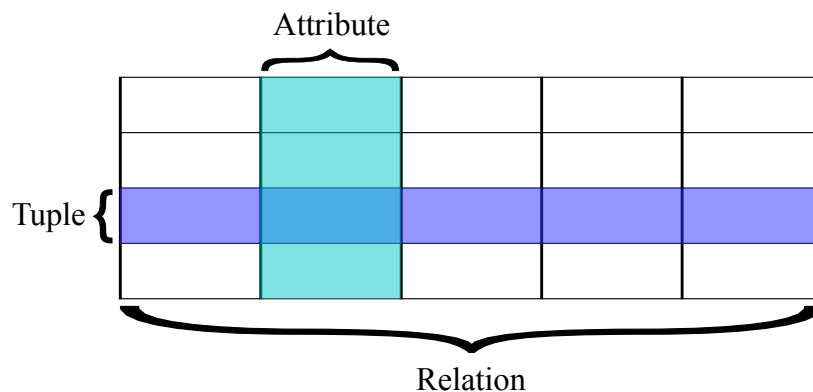


Figure 1.1: See [https://en.wikipedia.org/wiki/Relational\\_database](https://en.wikipedia.org/wiki/Relational_database).

## SQLite

The most common database management systems (DBMS) for relational databases are based on *Structured Query Language*, commonly called *SQL* (pronounced<sup>1</sup> “sequel”). Though SQL is a language in and of itself, most programming languages have tools for executing SQL routines. In Python, the most common variant of SQL is *SQLite*, implemented as the `sqlite3` module in the standard library.

A SQL database is stored in an external file, usually marked with the file extension `db` or `mdf`. These files should **not** be opened in Python with `open()` like text files; instead, any interactions with the database—creating, reading, updating, or deleting data—should occur as follows.

1. Create a connection to the database with `sqlite3.connect()`. This creates a database file if one does not already exist.
2. Get a *cursor*, an object that manages the actual traversal of the database, with the connection’s `cursor()` method.
3. Alter or read data with the cursor’s `execute()` method, which accepts an actual SQL command as a string.
4. Save any changes with the cursor’s `commit()` method, or revert changes with `rollback()`.
5. Close the connection.

```
>>> import sqlite3 as sql

# Establish a connection to a database file or create one if it doesn't exist.
>>> conn = sql.connect("my_database.db")
>>> try:
...     cur = conn.cursor()                # Get a cursor object.
...     cur.execute("SELECT * FROM MyTable") # Execute a SQL command.
... except sql.Error:                      # If there is an error,
...     conn.rollback()                    # revert the changes
...     raise                             # and raise the error.
... else:                                  # If there are no errors,
...     conn.commit()                      # save the changes.
... finally:
...     conn.close()                       # Close the connection.
```

### ACHTUNG!

Some changes, such as creating and deleting tables, are automatically committed to the database as part of the cursor’s `execute()` method. Be **extremely cautious** when deleting tables, as the action is immediate and permanent. Most changes, however, do not take effect in the database file until the connection’s `commit()` method is called. Be careful not to close the connection before committing desired changes, or those changes will not be recorded.

<sup>1</sup>See <https://english.stackexchange.com/questions/7231/how-is-sql-pronounced> for a brief history of the somewhat controversial pronunciation of SQL.

The `with` statement can be used with `open()` so that file streams are automatically closed, even in the event of an error. Likewise, combining the `with` statement with `sql.connect()` automatically rolls back changes if there is an error and commits them otherwise. However, the actual database connection is **not** closed automatically. With this strategy, the previous code block can be reduced to the following.

```
>>> try:
...     with sql.connect("my_database.db") as conn:
...         cur = conn.cursor()           # Get the cursor.
...         cur.execute("SELECT * FROM MyTable") # Execute a SQL command.
...     finally:                          # Commit or revert, then
...         conn.close()                  # close the connection.
```

## Managing Database Tables

SQLite uses five native data types (relatively few compared to other SQL systems) that correspond neatly to native Python data types.

Python Type	SQLite Type
<code>None</code>	<code>NULL</code>
<code>int</code>	<code>INTEGER</code>
<code>float</code>	<code>REAL</code>
<code>str</code>	<code>TEXT</code>
<code>bytes</code>	<code>BLOB</code>

The `CREATE TABLE` command, together with a table name and a schema, adds a new table to a database. The schema is a comma-separated list where each entry specifies the column name, the column data type,<sup>2</sup> and other optional parameters. For example, the following code adds a table called `MyTable` with the schema (`Name`, `ID`, `Age`) with appropriate data types.

```
>>> with sql.connect("my_database.db") as conn:
...     cur = conn.cursor()
...     cur.execute("CREATE TABLE MyTable (Name TEXT, ID INTEGER, Age REAL)")
...
>>> conn.close()
```

The `DROP TABLE` command deletes a table. However, using `CREATE TABLE` to try to create a table that already exists or using `DROP TABLE` to remove a nonexistent table raises an error. Use `DROP TABLE IF EXISTS` to remove a table without raising an error if the table doesn't exist. See Table 1.1 for more table management commands.

<sup>2</sup>Though SQLite does not force the data in a single column to be of the same type, most other SQL systems enforce uniform column types, so it is good practice to specify data types in the schema.

Operation	SQLite Command
Create a new table	<code>CREATE TABLE &lt;table&gt; (&lt;schema&gt;);</code>
Delete a table	<code>DROP TABLE &lt;table&gt;;</code>
Delete a table if it exists	<code>DROP TABLE IF EXISTS &lt;table&gt;;</code>
Add a new column to a table	<code>ALTER TABLE &lt;table&gt; ADD &lt;column&gt; &lt;dtype&gt;</code>
Remove an existing column	<code>ALTER TABLE &lt;table&gt; DROP COLUMN &lt;column&gt;;</code>
Rename an existing column	<code>ALTER TABLE &lt;table&gt; ALTER COLUMN &lt;column&gt; &lt;dtype&gt;;</code>

Table 1.1: SQLite commands for managing tables and columns.

**NOTE**

SQL commands like `CREATE TABLE` are often written in all caps to distinguish them from other parts of the query, like the table name. This is only a matter of style: SQLite, along with most other versions of SQL, is case insensitive. In Python's SQLite interface, the trailing semicolon is also unnecessary. However, most other database systems require it, so it's good practice to include the semicolon in Python.

**Problem 1.** Write a function that accepts the name of a database file. Connect to the database (and create it if it doesn't exist). Drop the tables `MajorInfo`, `CourseInfo`, `StudentInfo`, and `StudentGrades` from the database **if** they exist. Next, add the following tables to the database with the specified column names and types.

- `MajorInfo`: `MajorID` (integers) and `MajorName` (strings).
- `CourseInfo`: `CourseID` (integers) and `CourseName` (strings).
- `StudentInfo`: `StudentID` (integers), `StudentName` (strings), and `MajorID` (integers).
- `StudentGrades`: `StudentID` (integers), `CourseID` (integers), and `Grade` (strings).

Remember to commit and close the database. You should be able to execute your function more than once with the same input without raising an error.

To check the database, use the following commands to get the column names of a specified table. Assume here that the database file is called `students.db`.

```
>>> with sql.connect("students.db") as conn:
...     cur = conn.cursor()
...     cur.execute("SELECT * FROM StudentInfo;")
...     print([d[0] for d in cur.description])
...
['StudentID', 'StudentName', 'MajorID']
```

## Inserting, Removing, and Altering Data

Tuples are added to SQLite database tables with the `INSERT INTO` command.

```
# Add the tuple (Samuel Clemens, 1910421, 74.4) to MyTable in my_database.db.
>>> with sql.connect("my_database.db") as conn:
...     cur = conn.cursor()
...     cur.execute("INSERT INTO MyTable "
...                 "VALUES('Samuel Clemens', 1910421, 74.4);")
```

With this syntax, SQLite assumes that values match sequentially with the schema of the table. The schema of the table can also be written explicitly for clarity.

```
>>> with sql.connect("my_database.db") as conn:
...     cur = conn.cursor()
...     cur.execute("INSERT INTO MyTable(Name, ID, Age) "
...                 "VALUES('Samuel Clemens', 1910421, 74.4);")
```

### ACHTUNG!

**Never** use Python's string operations to construct a SQL query from variables. Doing so makes the program susceptible to a *SQL injection attack*.<sup>a</sup> Instead, use parameter substitution to construct dynamic commands: use a `?` character within the command, then provide the sequence of values as a second argument to `execute()`.

```
>>> with sql.connect("my_database.db") as conn:
...     cur = conn.cursor()
...     values = ('Samuel Clemens', 1910421, 74.4)
...     # Don't piece the command together with string operations!
...     # cur.execute("INSERT INTO MyTable VALUES " + str(values)) # BAD!
...     # Instead, use parameter substitution.
...     cur.execute("INSERT INTO MyTable VALUES(?,?,?);", values) # Good.
```

<sup>a</sup>See <https://xkcd.com/327/> for an example.

To insert several rows at a time to the same table, use the cursor object's `executemany()` method and parameter substitution with a list of tuples. This is typically much faster than using `execute()` repeatedly.

```
# Insert (Samuel Clemens, 1910421, 74.4) and (Jane Doe, 123, 20) to MyTable.
>>> with sql.connect("my_database.db") as conn:
...     cur = conn.cursor()
...     rows = [('John Smith', 456, 40.5), ('Jane Doe', 123, 20)]
...     cur.executemany("INSERT INTO MyTable VALUES(?,?,?);", rows)
```

**Problem 2.** Expand your function from Problem 1 so that it populates the tables with the data given in Tables 1.2a–1.2d.

MajorID	MajorName
1	Math
2	Science
3	Writing
4	Art

(a) MajorInfo

CourseID	CourseName
1	Calculus
2	English
3	Pottery
4	History

(b) CourseInfo

StudentID	StudentName	MajorID
401767594	Michelle Fernandez	1
678665086	Gilbert Chapman	NULL
553725811	Roberta Cook	2
886308195	Rene Cross	3
103066521	Cameron Kim	4
821568627	Mercedes Hall	NULL
206208438	Kristopher Tran	2
341324754	Cassandra Holland	1
262019426	Alfonso Phelps	NULL
622665098	Sammy Burke	2

(c) StudentInfo

StudentID	CourseID	Grade
401767594	4	C
401767594	3	B–
678665086	4	A+
678665086	3	A+
553725811	2	C
678665086	1	B
886308195	1	A
103066521	2	C
103066521	3	C–
821568627	4	D
821568627	2	A+
821568627	1	B
206208438	2	A
206208438	1	C+
341324754	2	D–
341324754	1	A–
103066521	4	A
262019426	2	B
262019426	3	C
622665098	1	A
622665098	2	A–

(d) StudentGrades

Table 1.2: Student database.

The `StudentInfo` and `StudentGrades` tables are also recorded in `student_info.csv` and `student_grades.csv`, respectively, with `NULL` values represented as `–1`. A CSV (comma-separated values) file can be read like a normal text file or with the `csv` module.

```
>>> import csv
>>> with open("student_info.csv", 'r') as infile:
...     rows = list(csv.reader(infile))
```

To validate your database, use the following command to retrieve the rows from a table.

```
>>> with sql.connect("students.db") as conn:
...     cur = conn.cursor()
...     for row in cur.execute("SELECT * FROM MajorInfo;"):
...         print(row)
(1, 'Math')
(2, 'Science')
(3, 'Writing')
(4, 'Art')
```



**Problem 3.** The data file `us_earthquakes.csv`<sup>a</sup> contains data from about 3,500 earthquakes in the United States since the 1769. Each row records the year, month, day, hour, minute, second, latitude, longitude, and magnitude of a single earthquake (in that order). Note that latitude, longitude, and magnitude are floats, while the remaining columns are integers.

Write a function that accepts the name of a database file. Drop the table `USEarthquakes` if it already exists, then create a new `USEarthquakes` table with schema (`Year`, `Month`, `Day`, `Hour`, `Minute`, `Second`, `Latitude`, `Longitude`, `Magnitude`). Populate the table with the data from `us_earthquakes.csv`. Remember to commit the changes and close the connection. (Hint: using `executemany()` is much faster than using `execute()` in a loop.)

<sup>a</sup>Retrieved from <https://datarepository.wolframcloud.com/resources/Sample-Data-US-Earthquakes>.

## The WHERE Clause

Deleting or altering existing data in a database requires some searching for the desired row or rows. The **WHERE** clause is a *predicate* that filters the rows based on a boolean condition. The operators `==`, `!=`, `<`, `>`, `<=`, `>=`, **AND**, **OR**, and **NOT** all work as expected to create search conditions.

```
>>> with sql.connect("my_database.db") as conn:
...     cur = conn.cursor()
...     # Delete any rows where the Age column has a value less than 30.
...     cur.execute("DELETE FROM MyTable WHERE Age < 30;")
...     # Change the Name of "Samuel Clemens" to "Mark Twain".
...     cur.execute("UPDATE MyTable SET Name='Mark Twain' WHERE ID==1910421;")
```

If the **WHERE** clause were omitted from either of the previous commands, every record in `MyTable` would be affected. **Always** use a very specific **WHERE** clause when removing or updating data.

Operation	SQLite Command
Add a new row to a table	<code>INSERT INTO table VALUES(&lt;values&gt;);</code>
Remove rows from a table	<code>DELETE FROM &lt;table&gt; WHERE &lt;condition&gt;;</code>
Change values in existing rows	<code>UPDATE &lt;table&gt; SET &lt;column1&gt;=&lt;value1&gt;, ... WHERE &lt;condition&gt;;</code>

Table 1.3: SQLite commands for inserting, removing, and updating rows.

**Problem 4.** Modify your function from Problems 1 and 2 so that in the `StudentInfo` table, values of `-1` in the `MajorID` column are replaced with **NULL** values.

Also modify your function from Problem 3 in the following ways.

1. Remove rows from `USEarthquakes` that have a value of 0 for the `Magnitude`.
2. Replace 0 values in the `Day`, `Hour`, `Minute`, and `Second` columns with **NULL** values.

## Reading and Analyzing Data

Constructing and managing databases is fundamental, but most time in SQL is spent analyzing existing data. A *query* is a SQL command that reads all or part of a database without actually modifying the data. Queries start with the **SELECT** command, followed by column and table names and additional (optional) conditions. The results of a query, called the *result set*, are accessed through the cursor object. After calling `execute()` with a SQL query, use `fetchall()` or another cursor method from Table 1.4 to get the list of matching tuples.

Method	Description
<code>execute()</code>	Execute a single SQL command
<code>executemany()</code>	Execute a single SQL command over different values
<code>executescript()</code>	Execute a SQL script (multiple SQL commands)
<code>fetchone()</code>	Return a single tuple from the result set
<code>fetchmany(n)</code>	Return the next <i>n</i> rows from the result set as a list of tuples
<code>fetchall()</code>	Return the entire result set as a list of tuples

Table 1.4: Methods of database cursor objects.

```
>>> conn = sql.connect("students.db")
>>> cur = conn.cursor()

# Get tuples of the form (StudentID, StudentName) from the StudentInfo table.
>>> cur.execute("SELECT StudentID, StudentName FROM StudentInfo;")
>>> cur.fetchone()          # List the first match (a tuple).
(401767594, 'Michelle Fernandez')

>>> cur.fetchmany(3)         # List the next three matches (a list of tuples).
[(678665086, 'Gilbert Chapman'),
 (553725811, 'Roberta Cook'),
 (886308195, 'Rene Cross')]

>>> cur.fetchall()          # List the remaining matches.
[(103066521, 'Cameron Kim'),
 (821568627, 'Mercedes Hall'),
 (206208438, 'Kristopher Tran'),
 (341324754, 'Cassandra Holland'),
 (262019426, 'Alfonso Phelps'),
 (622665098, 'Sammy Burke')]

# Use * in place of column names to get all of the columns.
>>> cur.execute("SELECT * FROM MajorInfo;").fetchall()
[(1, 'Math'), (2, 'Science'), (3, 'Writing'), (4, 'Art')]

>>> conn.close()
```

The **WHERE** predicate can also refine a **SELECT** command. If the condition depends on a column in a different table from the data that is being a selected, create a *table alias* with the **AS** command to specify columns in the form `table.column`.

```

>>> conn = sql.connect("students.db")
>>> cur = conn.cursor()

# Get the names of all math majors.
>>> cur.execute("SELECT SI.StudentName "
...             "FROM StudentInfo AS SI, MajorInfo AS MI "
...             "WHERE SI.MajorID == MI.MajorID AND MI.MajorName == 'Math'")
# The result set is a list of 1-tuples; extract the entry from each tuple.
>>> [t[0] for t in cur.fetchall()]
['Cassandra Holland', 'Michelle Fernandez']

# Get the names and grades of everyone in English class.
>>> cur.execute("SELECT SI.StudentName, SG.Grade "
...             "FROM StudentInfo AS SI, StudentGrades AS SG "
...             "WHERE SI.StudentID == SG.StudentID AND CourseID == 2;")
>>> cur.fetchall()
[('Roberta Cook', 'C'),
 ('Cameron Kim', 'C'),
 ('Mercedes Hall', 'A+'),
 ('Kristopher Tran', 'A'),
 ('Cassandra Holland', 'D-'),
 ('Alfonso Phelps', 'B'),
 ('Sammy Burke', 'A-')]

>>> conn.close()

```

**Problem 5.** Write a function that accepts the name of a database file. Assuming the database to be in the format of the one created in Problems 1 and 2, query the database for all tuples of the form (StudentName, CourseName) where that student has an “A” or “A+” grade in that course. Return the list of tuples.

## Aggregate Functions

A result set can be analyzed in Python using tools like NumPy, but SQL itself provides a few tools for computing a few very basic statistics: `AVG()`, `MIN()`, `MAX()`, `SUM()`, and `COUNT()` are *aggregate functions* that compress the columns of a result set into the desired quantity.

```

>>> conn = sql.connect("students.db")
>>> cur = conn.cursor()

# Get the number of students and the lowest ID number in StudentInfo.
>>> cur.execute("SELECT COUNT(StudentName), MIN(StudentID) FROM StudentInfo;")
>>> cur.fetchall()
[(10, 103066521)]

```

**Problem 6.** Write a function that accepts the name of a database file. Assuming the database to be in the format of the one created in Problem 3, query the `USEarthquakes` table for the following information.

- The magnitudes of the earthquakes during the 19th century (1800–1899).
- The magnitudes of the earthquakes during the 20th century (1900–1999).
- The average magnitude of all earthquakes in the database.

Create a single figure with two subplots: a histogram of the magnitudes of the earthquakes in the 19th century, and a histogram of the magnitudes of the earthquakes in the 20th century. Show the figure, then return the average magnitude of all of the earthquakes in the database. Be sure to return an actual number, not a list or a tuple.

(Hint: use `np.ravel()` to convert a result set of 1-tuples to a 1-D array.)

#### NOTE

Problem 6 raises an interesting question: are the number of earthquakes in the United States increasing with time, and if so, how drastically? A closer look shows that only 3 earthquakes were recorded (in this data set) from 1700–1799, 208 from 1800–1899, and a whopping 3049 from 1900–1999. Is the increase in earthquakes due to there actually being more earthquakes, or to the improvement of earthquake detection technology? The best answer without conducting additional research is “probably both.” Be careful to question the nature of your data—how it was gathered, what it may be lacking, what biases or lurking variables might be present—before jumping to strong conclusions.

See the following for more info on the `sqlite3` and SQL in general.

- <https://docs.python.org/3/library/sqlite3.html>
- <https://www.w3schools.com/sql/>
- [https://en.wikipedia.org/wiki/SQL\\_injection](https://en.wikipedia.org/wiki/SQL_injection)

## Additional Material

### Shortcuts for WHERE Conditions

Complicated **WHERE** conditions can be simplified with the following commands.

- **IN**: check for equality to one of several values quickly, similar to Python's **in** operator. In other words, the following SQL commands are equivalent.

```
SELECT * FROM StudentInfo WHERE MajorID == 1 OR MajorID == 2;  
SELECT * FROM StudentInfo WHERE MajorID IN (1,2);
```

- **BETWEEN**: check two (inclusive) inequalities quickly. The following are equivalent.

```
SELECT * FROM MyTable WHERE AGE >= 20 AND AGE <= 60;  
SELECT * FROM MyTable WHERE AGE BETWEEN 20 AND 60;
```



# 2

## SQL 2 (The Sequel)

**Lab Objective:** *Since SQL databases contain multiple tables, retrieving information about the data can be complicated. In this lab we discuss joins, grouping, and other advanced SQL query concepts to facilitate rapid data retrieval.*

We will use the following database as an example throughout this lab, found in `students.db`.

MajorID	MajorName
1	Math
2	Science
3	Writing
4	Art

(a) MajorInfo

CourseID	CourseName
1	Calculus
2	English
3	Pottery
4	History

(b) CourseInfo

StudentID	StudentName	MajorID
401767594	Michelle Fernandez	1
678665086	Gilbert Chapman	NULL
553725811	Roberta Cook	2
886308195	Rene Cross	3
103066521	Cameron Kim	4
821568627	Mercedes Hall	NULL
206208438	Kristopher Tran	2
341324754	Cassandra Holland	1
262019426	Alfonso Phelps	NULL
622665098	Sammy Burke	2

(c) StudentInfo

StudentID	CourseID	Grade
401767594	4	C
401767594	3	B-
678665086	4	A+
678665086	3	A+
553725811	2	C
678665086	1	B
886308195	1	A
103066521	2	C
103066521	3	C-
821568627	4	D
821568627	2	A+
821568627	1	B
206208438	2	A
206208438	1	C+
341324754	2	D-
341324754	1	A-
103066521	4	A
262019426	2	B
262019426	3	C
622665098	1	A
622665098	2	A-

(d) StudentGrades

Table 2.1: Student database.

### Joining Tables

A *join* combines rows from different tables in a database based on common attributes. In other words, a join operation creates a new, temporary table containing data from 2 or more existing tables. Join commands in SQLite have the following general syntax.

```
SELECT <alias.column, ...>
  FROM <table> AS <alias> JOIN <table> AS <alias>, ...
 ON <alias.column> == <alias.column>, ...
 WHERE <condition>
```

The **ON** clause tells the query how to join tables together. Typically if there are  $N$  tables being joined together, there should be  $N - 1$  conditions in the **ON** clause.

## Inner Joins

An *inner join* creates a temporary table with the rows that have exact matches on the attribute(s) specified in the **ON** clause. Inner joins **intersect** two or more tables, as in Figure 2.1a.

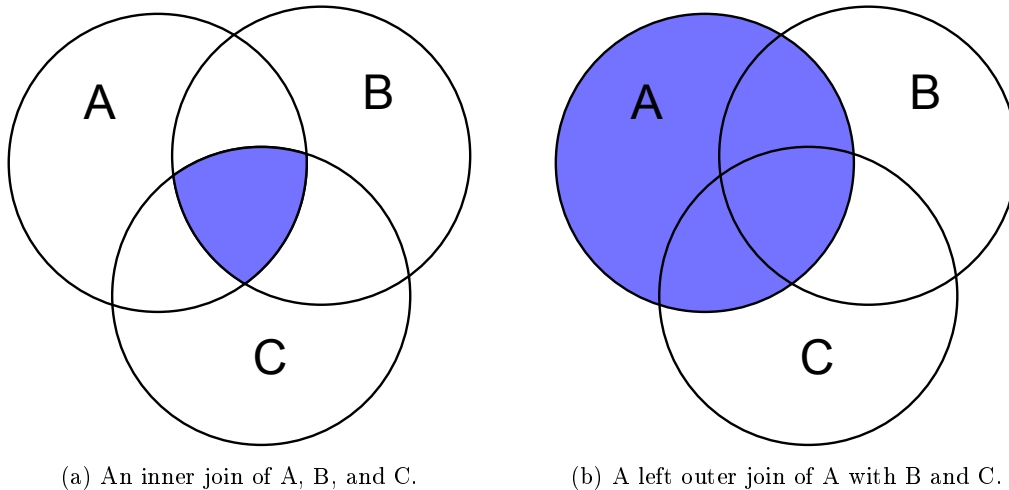


Figure 2.1

For example, Table 2.1c (**StudentInfo**) and Table 2.1a (**MajorInfo**) both have a **MajorID** column, so the tables can be joined by pairing rows that have the same **MajorID**. Such a join temporarily creates the following table.

StudentID	StudentName	MajorID	MajorID	MajorName
401767594	Michelle Fernandez	1	1	Math
553725811	Roberta Cook	2	2	Science
886308195	Rene Cross	3	3	Writing
103066521	Cameron Kim	4	4	Art
206208438	Kristopher Tran	2	2	Science
341324754	Cassandra Holland	1	1	Math
622665098	Sammy Burke	2	2	Science

Table 2.2: An inner join of **StudentInfo** and **MajorInfo** on **MajorID**.

Notice that this table is missing the rows where **MajorID** was **NULL** in the **StudentInfo** table. This is because there was no match for **NULL** in the **MajorID** column of the **MajorInfo** table, so the inner join throws those rows away.



Because joins deal with multiple tables at once, it is important to assign table aliases with the `AS` command. Join statements can also be supplemented with `WHERE` clauses like regular queries.

```
>>> import sqlite3 as sql
>>> conn = sql.connect("students.db")
>>> cur = conn.cursor()

>>> cur.execute("SELECT * "
...             "FROM StudentInfo AS SI INNER JOIN MajorInfo AS MI "
...             "ON SI.MajorID = MI.MajorID;").fetchall()
[(401767594, 'Michelle Fernandez', 1, 1, 'Math'),
 (553725811, 'Roberta Cook', 2, 2, 'Science'),
 (886308195, 'Rene Cross', 3, 3, 'Writing'),
 (103066521, 'Cameron Kim', 4, 4, 'Art'),
 (206208438, 'Kristopher Tran', 2, 2, 'Science'),
 (341324754, 'Cassandra Holland', 1, 1, 'Math'),
 (622665098, 'Sammy Burke', 2, 2, 'Science')]

# Select the names and ID numbers of the math majors.
>>> cur.execute("SELECT SI.StudentName, SI.StudentID "
...             "FROM StudentInfo AS SI INNER JOIN MajorInfo AS MI "
...             "ON SI.MajorID = MI.MajorID "
...             "WHERE MI.MajorName == 'Math;").fetchall()
[('Cassandra Holland', 341324754), ('Michelle Fernandez', 401767594)]
```

**Problem 1.** Write a function that accepts the name of a database file. Assuming the database to be in the format of Tables 2.1a–2.1d, query the database for the list of the names of students who have a B grade in any course (not a B– or a B+).

## Outer Joins

A *left outer join*, sometimes called a *left join*, creates a temporary table with **all** of the rows from the first (left-most) table, and all the “matched” rows on the given attribute(s) from the other relations. Rows from the left table that don’t match up with the columns from the other tables are supplemented with `NULL` values to fill extra columns. Compare the following table and code to Table 2.2.

StudentID	StudentName	MajorID	MajorID	MajorName
401767594	Michelle Fernandez	1	1	Math
678665086	Gilbert Chapman	NULL	NULL	NULL
553725811	Roberta Cook	2	2	Science
886308195	Rene Cross	3	3	Writing
103066521	Cameron Kim	4	4	Art
821568627	Mercedes Hall	NULL	NULL	NULL
206208438	Kristopher Tran	2	2	Science
341324754	Cassandra Holland	1	1	Math
262019426	Alfonso Phelps	NULL	NULL	NULL
622665098	Sammy Burke	2	2	Science

Table 2.3: A left outer join of `StudentInfo` and `MajorInfo` on `MajorID`.

```
>>> cur.execute("SELECT * "
...             "FROM StudentInfo AS SI LEFT OUTER JOIN MajorInfo AS MI "
...             "ON SI.MajorID = MI.MajorID;").fetchall()
[(401767594, 'Michelle Fernandez', 1, 1, 'Math'),
 (678665086, 'Gilbert Chapman', None, None, None),
 (553725811, 'Roberta Cook', 2, 2, 'Science'),
 (886308195, 'Rene Cross', 3, 3, 'Writing'),
 (103066521, 'Cameron Kim', 4, 4, 'Art'),
 (821568627, 'Mercedes Hall', None, None, None),
 (206208438, 'Kristopher Tran', 2, 2, 'Science'),
 (341324754, 'Cassandra Holland', 1, 1, 'Math'),
 (262019426, 'Alfonso Phelps', None, None, None),
 (622665098, 'Sammy Burke', 2, 2, 'Science')]
```

Some flavors of SQL also support the **RIGHT OUTER JOIN** command, but `sqlite3` does not recognize the command since `T1 RIGHT OUTER JOIN T2` is equivalent to `T2 LEFT OUTER JOIN T1`.

## Joining Multiple Tables

Complicated queries often join several different relations. If the same kind join is being used, the relations and conditional statements can be put in list form. For example, the following code selects courses that Kristopher Tran has taken, and the grades that he got in those courses, by joining three tables together. Note that 2 conditions are required in the **ON** clause in this case.

```
>>> cur.execute("SELECT CI.CourseName, SG.Grade "
...             "FROM StudentInfo AS SI "           # Join 3 tables.
...             "INNER JOIN CourseInfo AS CI, StudentGrades SG "
...             "ON SI.StudentID==SG.StudentID AND CI.CourseID==SG.CourseID "
...             "WHERE SI.StudentName == 'Kristopher Tran';").fetchall()
[('Calculus', 'C+'), ('English', 'A')]
```

To use different kinds of joins in a single query, append one join statement after another. The join closest to the beginning of the statement is executed first, creating a temporary table, and the next join attempts to operate on that table. The following example performs an additional join on Table 2.3 to find the name and major of every student who got a C in a class.

```
# Do an inner join on the results of the left outer join.
>>> cur.execute("SELECT SI.StudentName, MI.MajorName "
...             "FROM StudentInfo AS SI LEFT OUTER JOIN MajorInfo AS MI "
...             "ON SI.MajorID == MI.MajorID "
...             "INNER JOIN StudentGrades AS SG "
...             "ON SI.StudentID = SG.StudentID "
...             "WHERE SG.Grade = 'C';").fetchall()
[('Michelle Fernandez', 'Math'),
 ('Roberta Cook', 'Science'),
 ('Cameron Kim', 'Art'),
 ('Alfonso Phelps', None)]
```

In this last example, note carefully that Alfonso Phelps would have been excluded from the result set if an inner join was performed first instead of an outer join (since he lacks a major).

**Problem 2.** Write a function that accepts the name of a database file. Query the database for all tuples of the form (Name, MajorName, Grade) where Name is a student's name and Grade is their grade in Calculus. Only include results for students that are actually taking Calculus, but be careful not to exclude students who haven't declared a major.

## Grouping Data

Many data sets can be naturally sorted into groups. The **GROUP BY** command gathers rows from a table and groups them by a certain attribute. The groups must be then combined by one of the *aggregate functions* **AVG()**, **MIN()**, **MAX()**, **SUM()**, or **COUNT()**. The following code groups the rows in Table 2.1d by **studentID** and counts the number of entries in each group.

```
>>> cur.execute("SELECT StudentID, COUNT(*) " # * means "all of the rows".
...             "FROM StudentGrades "
...             "GROUP BY StudentID").fetchall()
[(103066521, 3),
 (206208438, 2),
 (262019426, 2),
 (341324754, 2),
 (401767594, 2),
 (553725811, 1),
 (622665098, 2),
 (678665086, 3),
 (821568627, 3),
 (886308195, 1)]
```

**GROUP BY** can also be used in conjunction with joins. The join creates a temporary table like Tables 2.2 or 2.3, the results of which can then be grouped.

```
>>> cur.execute("SELECT SI.StudentName, COUNT(*) "
...             "FROM StudentGrades AS SG INNER JOIN StudentInfo AS SI "
...             "ON SG.StudentID == SI.StudentID "
...             "GROUP BY SG.StudentID").fetchall()
[('Cameron Kim', 3),
 ('Kristopher Tran', 2),
 ('Alfonso Phelps', 2),
 ('Cassandra Holland', 2),
 ('Michelle Fernandez', 2),
 ('Roberta Cook', 1),
 ('Sammy Burke', 2),
 ('Gilbert Chapman', 3),
 ('Mercedes Hall', 3),
 ('Rene Cross', 1)]
```

Just like the **WHERE** clause chooses rows in a relation, the **HAVING** clause chooses groups from the result of a **GROUP BY** based on some criteria related to the groupings. For this particular command, it is often useful (but not always necessary) to create an alias for the columns of the result set with the **AS** operator. For instance, the result set of the previous example can be filtered down to only contain students who are taking 3 courses.

```
>>> cur.execute("SELECT SI.StudentName, COUNT(*) as num_courses " # Alias.
...             "FROM StudentGrades AS SG INNER JOIN StudentInfo AS SI "
...             "ON SG.StudentID == SI.StudentID "
...             "GROUP BY SG.StudentID "
...             "HAVING num_courses == 3").fetchall() # Refer to alias later.
[('Cameron Kim', 3), ('Gilbert Chapman', 3), ('Mercedes Hall', 3)]

# Alternatively, get just the student names.
>>> cur.execute("SELECT SI.StudentName " # No alias.
...             "FROM StudentGrades AS SG INNER JOIN StudentInfo AS SI "
...             "ON SG.StudentID == SI.StudentID "
...             "GROUP BY SG.StudentID "
...             "HAVING COUNT(*) == 3").fetchall()
[('Cameron Kim',), ('Gilbert Chapman',), ('Mercedes Hall',)]
```

**Problem 3.** Write a function that accepts a database file. Query the database for the list of the names of courses that have at least 5 student enrolled in them.

## Other Miscellaneous Commands

### Ordering Result Sets

The **ORDER BY** command sorts a result set by one or more attributes. Sorting can be done in ascending or descending order with **ASC** or **DESC**, respectively. This is always the very last statement in a query.

```
>>> cur.execute("SELECT SI.StudentName, COUNT(*) AS num_courses " # Alias.
...             "FROM StudentGrades AS SG INNER JOIN StudentInfo AS SI "
...             "ON SG.StudentID == SI.StudentID "
...             "GROUP BY SG.StudentID "
...             "ORDER BY num_courses DESC, SI.StudentName ASC").fetchall()
[('Cameron Kim', 3), # The results are now ordered by the
('Gilbert Chapman', 3), # number of courses each student is in,
('Mercedes Hall', 3), # then alphabetically by student name.
('Alfonso Phelps', 2),
('Cassandra Holland', 2),
('Kristopher Tran', 2),
('Michelle Fernandez', 2),
('Sammy Burke', 2),
('Rene Cross', 1),
('Roberta Cook', 1)]
```

**Problem 4.** Write a function that accepts a database file. Query the given database for tuples of the form (MajorName, N) where N is the number of students in the specified major. Sort the results in ascending order by the count N.

## Searching Text with Wildcards

The **LIKE** operator within a **WHERE** clause matches patterns in a **TEXT** column. The special characters **%** and **\_** and called *wildcards* that match any number of characters or a single character, respectively. For instance, **%Z\_** matches any string of characters ending in a Z then another character, and **%i%** matches any string containing the letter i.

```
>>> results = cur.execute("SELECT StudentName FROM StudentInfo "
...                        "WHERE StudentName LIKE '%i%';").fetchall()
>>> [r[0] for r in results]
['Michelle Fernandez', 'Gilbert Chapman', 'Cameron Kim', 'Kristopher Tran']
```

**Problem 5.** Write a function that accepts a database file. Query the database for tuples of the form (StudentName, MajorName) where the last name of the specified student begins with the letter C.

## Case Expressions

A case expression maps the values in a column using boolean logic. There are two forms of a case expression: simple and searched. A *simple case expression* matches and replaces specified attributes.

```
# Replace the values MajorID with new custom values.
>>> cur.execute("SELECT StudentName, CASE MajorID "
...             "WHEN 1 THEN 'Mathematics' "
...             "WHEN 2 THEN 'Soft Science' "
...             "WHEN 3 THEN 'Writing and Editing' "
...             "WHEN 4 THEN 'Fine Arts' "
...             "ELSE 'Undeclared' END "
...             "FROM StudentInfo "
...             "ORDER BY StudentName ASC;").fetchall()
[('Alfonso Phelps', 'Undeclared'),
 ('Cameron Kim', 'Fine Arts'),
 ('Cassandra Holland', 'Mathematics'),
 ('Gilbert Chapman', 'Undeclared'),
 ('Kristopher Tran', 'Soft Science'),
 ('Mercedes Hall', 'Undeclared'),
 ('Michelle Fernandez', 'Mathematics'),
 ('Rene Cross', 'Writing and Editing'),
 ('Roberta Cook', 'Soft Science'),
 ('Sammy Burke', 'Soft Science')]
```

A *searched case expression* involves using a boolean expression at each step, instead of listing all of the possible values for an attribute.

```
# Change NULL values in MajorID to 'Undeclared' and non-NULL to 'Declared'.
>>> cur.execute("SELECT StudentName, CASE "
...             "WHEN MajorID IS NULL THEN 'Undeclared' "
...             "ELSE 'Declared' END "
...             "FROM StudentInfo "
...             "ORDER BY StudentName ASC;").fetchall()
[('Alfonso Phelps', 'Undeclared'),
 ('Cameron Kim', 'Declared'),
 ('Cassandra Holland', 'Declared'),
 ('Gilbert Chapman', 'Undeclared'),
 ('Kristopher Tran', 'Declared'),
 ('Mercedes Hall', 'Undeclared'),
 ('Michelle Fernandez', 'Declared'),
 ('Rene Cross', 'Declared'),
 ('Roberta Cook', 'Declared'),
 ('Sammy Burke', 'Declared')]
```

## Chaining Queries

The result set of any SQL query is really just another table with data from the original database. Separate queries can be made from result sets by enclosing the entire query in parentheses. For these sorts of operations, it is very important to carefully label the columns resulting from a subquery.

```
>>> cur.execute("SELECT majorstatus, COUNT(*) AS majorcount "
...             "FROM ( "                                     # Begin subquery.
...             "SELECT StudentName, CASE "
...             "WHEN MajorID IS NULL THEN 'Undeclared' "
...             "ELSE 'Declared' END AS majorstatus "
...             "FROM StudentInfo) "                         # End subquery.
...             "GROUP BY majorstatus "
...             "ORDER BY majorcount DESC;").fetchall()
[('Declared', 7), ('Undeclared', 3)]
```

**Problem 6.** Write a function that accepts the name of a database file. Query the database for tuples of the form (StudentName, N, GPA) where N is the number of courses that the specified student is enrolled in and GPA is their grade point average based on the following point system.

A+, A = 4.0	B = 3.0	C = 2.0	D = 1.0
A- = 3.7	B- = 2.7	C- = 1.7	D- = 0.7
B+ = 3.4	C+ = 2.4	D+ = 1.4	

Order the results from greatest GPA to least.

# 3

## Regular Expressions

**Lab Objective:** *Cleaning and formatting data are fundamental problems in data science. Regular expressions are an important tool for working with text carefully and efficiently, and are useful for both gathering and cleaning data. This lab introduces regular expression syntax and common practices, including an application to a data cleaning problem.*

A *regular expression* or *regex* is a string of characters that follows a certain syntax to specify a pattern. Strings that follow the pattern are said to *match* the expression (and vice versa). A single regular expression can match a large set of strings, such as the set of all valid email addresses.

### ACHTUNG!

There are some universal standards for regular expression syntax, but the exact syntax varies slightly depending on the program or language. However, the syntax presented in this lab (for Python) is sufficiently similar to any other regex system. Consider learning to use regular expressions in Vim or your favorite text editor, keeping in mind that there will be slight syntactic differences from what is presented here.

## Regular Expression Syntax in Python

The `re` module implements regular expressions in Python. The function `re.compile()` takes in a regular expression string and returns a corresponding *pattern* object, which has methods for determining if and how other strings match the pattern. For example, the `search()` method returns `None` for a string that doesn't match, and a *match* object for a string that does.

Note the `match()` method for pattern objects only matches strings that satisfy the pattern **at the beginning** of the string. To answer the question “does any part of my target string match this regular expression?” always use the `search()` method.

```
>>> import re
>>> pattern = re.compile("cat")      # Make a pattern for finding 'cat'.
>>> bool(pattern.search("cat"))      # 'cat' matches 'cat', of course.
True
>>> bool(pattern.match("catfish"))  # 'catfish' starts with 'cat'.
```

```
True
>>> bool(pattern.match("fishcat")) # 'fishcat' doesn't start with 'cat'.
False
>>> bool(pattern.search("fishcat")) # but it does contain 'cat'.
True
>>> bool(pattern.search("hat"))      # 'hat' does not contain 'cat'.
False
```

Most of the functions in the `re` module are shortcuts for compiling a pattern object and calling one of its methods. Using `re.compile()` is good practice because the resulting object is reusable, while each call to `re.search()` compiles a new (but redundant) pattern object. For example, the following lines of code are equivalent.

```
>>> bool(re.compile("cat").search("catfish"))
True
>>> bool(re.search("cat", "catfish"))
True
```

**Problem 1.** Write a function that compiles and returns a regular expression pattern object with the pattern string `"python"`.

## Literal Characters and Metacharacters

The following string characters (separated by spaces) are *metacharacters* in Python's regular expressions, meaning they have special significance in a pattern string: `. ^ $ * + ? { } [ ] \ | ( )`.

A regular expression that matches strings with one or more metacharacters requires two things.

1. Use *raw strings* instead of regular Python strings by prefacing the string with an `r`, such as `r"cat"`. The resulting string interprets backslashes as actual backslash characters, rather than the start of an escape sequence like `\n` or `\t`.
2. Preface any metacharacters with a backslash to indicate a literal character. For example, to match the string `"$3.99? Thanks."`, use `r"\$3\.99? Thanks\"`.

Without raw strings, every backslash in has to be written as a double backslash, which makes many regular expression patterns hard to read (`"\\$3\\.99\\? Thanks\\"`).

**Problem 2.** Write a function that compiles and returns a regular expression pattern object that matches the string `"^@{0}?(?)[%]{.}(*)[_]{&}$"`.

The regular expressions of Problems 1 and 2 only match strings that are or include the exact pattern. The metacharacters allow regular expressions to have much more flexibility and control so that a single pattern can match a wide variety of strings, or a very specific set of strings. The *line anchor* metacharacters `^` and `$` are used to match the **start** and the **end** of a line of text, respectively. This shrinks the matching set, even when using the `search()` method instead of the `match()` method. For example, the only single-line string that the expression `^x$` matches is `'x'`, whereas the expression `'x'` can match any string with an `'x'` in it.



The *pipe* character `|` is a logical OR in a regular expression: `A|B` matches `A` or `B`. The parentheses `()` create a *group* in a regular expression. A group establishes an order of operations in an expression. For example, in the regex `"^one|two fish$"`, precedence is given to the invisible string concatenation between `"two"` and `"fish"`, while `"^(one|two) fish$"` gives precedence to the `'|'` metacharacter.

```
>>> fish = re.compile(r"^(one|two) fish$")
>>> for test in ["one fish", "two fish", "red fish", "one two fish"]:
...     print(test + ': ', bool(fish.search(test)))
...
one fish: True
two fish: True
red fish: False
one two fish: False
```

**Problem 3.** Write a function that compiles and returns a regular expression pattern object that matches the following strings, and no other strings, even with `re.search()`.

```
"Book store"      "Mattress store"      "Grocery store"
"Book supplier"   "Mattress supplier"   "Grocery supplier"
```

## Character Classes

The hard bracket metacharacters `[` and `]` are used to create *character classes*, a part of a regular expression that can match a variety of characters. For example, the pattern `[abc]` matches any of the characters `a`, `b`, or `c`. This is different than a group delimited by parentheses: a group can match multiple characters, while a character class matches only one character. For instance, `[abc]` does not match `ab` or `abc`, and `(abc)` matches `abc` but not `ab` or even `a`.

Within character classes, there are two additional metacharacters. When `^` appears as the **first character** in a character class, right after the opening bracket `[`, the character class matches anything **not** specified instead. In other words, `^` is the set complement operation on the character class. Additionally, the dash `-` specifies a range of values. For instance, `[0-9]` matches any digit, and `[a-z]` matches any lowercase letter. Thus `[^0-9]` matches any character **except** for a digit, and `[^a-z]` matches any character **except** for lowercase letters. Keep in mind that the dash `-`, when at the beginning or end of the character class, will match the literal `'-'`.

```
>>> p1, p2 = re.compile(r"^[a-z][^0-7]$"), re.compile(r"^[^abcA-C][0-27-9]$")
>>> for test in ["d8", "aa", "E9", "EE", "d88"]:
...     print(test + ': ', bool(p1.search(test)), bool(p2.search(test)))
...
d8: True True
aa: True False          # a is not in [^abcA-C] or [0-27-9].
E9: False True          # E is not in [a-z].
EE: False False         # E is not in [a-z] or [0-27-9].
d88: False False        # Too many characters.
```

Note that `[0-27-9]` acts like `[(0-2)|(7-9)]`.

There are also a variety of shortcuts that represent common character classes, listed in Table 3.1. Familiarity with these shortcuts makes some regular expressions significantly more readable.

Character	Description
\b	Matches the empty string, but only at the start or end of a word.
\s	Matches any whitespace character; equivalent to <code>[ \t\n\r\f\v]</code> .
\S	Matches any non-whitespace character; equivalent to <code>[^\s]</code> .
\d	Matches any decimal digit; equivalent to <code>[0-9]</code> .
\D	Matches any non-digit character; equivalent to <code>[^\d]</code> .
\w	Matches any alphanumeric character; equivalent to <code>[a-zA-Z0-9_]</code> .
\W	Matches any non-alphanumeric character; equivalent to <code>[^\w]</code> .

Table 3.1: Character class shortcuts.

Any of the character class shortcuts can be used within other custom character classes. For example, `[_A-Z\s]` matches an underscore, capital letter, or whitespace character.

Finally, a period `.` matches **any** character except for a line break. This is a very powerful metacharacter; be careful to only use it when part of the regular expression really should match **any** character.

```
# Match any three-character string with a digit in the middle.
>>> pattern = re.compile(r"^.\d.$")
>>> for test in ["a0b", "888", "n2%", "abc", "cat"]:
...     print(test + ': ', bool(pattern.search(test)))
...
a0b: True
888: True
n2%: True
abc: False
cat: False

# Match two letters followed by a number and two non-newline characters.
>>> pattern = re.compile(r"^[a-zA-Z][a-zA-Z]\d..$")
>>> for test in ["tk421", "bb8!?", "JB007", "Boba?"]:
...     print(test + ': ', bool(pattern.search(test)))
...
tk421: True
bb8!?: True
JB007: True
Boba?: False
```

The following table is a useful recap of some common regular expression metacharacters.

Character	Description
.	Matches any character except a newline.
^	Matches the start of the string.
\$	Matches the end of the string or just before the newline at the end of the string.
	A B creates an regular expression that will match either A or B.
[...]	Indicates a set of characters. A ^ as the first character indicates a complementing set.
(...)	Matches the regular expression inside the parentheses. The contents can be retrieved or matched later in the string.

Table 3.2: Standard regular expression metacharacters in Python.

## Repetition

The remaining metacharacters are for matching a specified number of characters. This allows a single regular expression to match strings of varying lengths.

Character	Description
*	Matches 0 or more repetitions of the preceding regular expression.
+	Matches 1 or more repetitions of the preceding regular expression.
?	Matches 0 or 1 of the preceding regular expression.
{m,n}	Matches from m to n repetitions of the preceding regular expression.
*, +, ?, {m,n}?	Non-greedy versions of the previous four special characters.

Table 3.3: Repetition metacharacters for regular expressions in Python.

Each of the repetition operators acts on the expression immediately preceding it. This could be a single character, a group, or a character class. For instance, `(abc)+` matches `abc`, `abcabc`, `abcabcabc`, and so on, but not `aba` or `cba`. On the other hand, `[abc]*` matches any sequence of `a`, `b`, and `c`, including `abcabc` and `aabbcc`.

The curly braces `{}` specify a custom number of repetitions allowed. `{,n}` matches **up to** `n` instances, `{m,}` matches **at least** `m` instances, `{k}` matches **exactly** `k` instances, and `{m,n}` matches from `m` to `n` instances. Thus the `?` operator is equivalent to `{,1}` and `+` is equivalent to `{1,}`.

```
# Match exactly 3 'a' characters.
>>> pattern = re.compile(r"^a{3}$")
>>> for test in ["aa", "aaa", "aaaa", "aba"]:
...     print(test + ': ', bool(pattern.search(test)))
...
aa: False                                # Too few.
aaa: True
aaaa: False                              # Too many.
aba: False
```

Be aware that line anchors are especially important when using repetition operators. Consider the following (bad) example and compare it to the previous example.

```
# Match exactly 3 'a' characters, hopefully.
>>> pattern = re.compile(r"a{3}")
>>> for test in ["aaa", "aaaa", "aaaaa", "aaaab"]:
```

```
...     print(test + ': ', bool(pattern.search(test)))
...
aaa: True
aaaa: True           # Should be too many!
aaaaa: True          # Should be too many!
aaaab: True          # Too many, and even with the 'b'?
```

The unexpected matches occur because `"aaa"` is at the beginning of each of the test strings. With the line anchors `^` and `$`, the search truly only matches the exact string `"aaa"`.

**Problem 4.** A *valid Python identifier* (a valid variable name) is any string starting with an alphabetic character or an underscore, followed by any (possibly empty) sequence of alphanumeric characters and underscores.

A *valid python parameter definition* is defined as the concatenation of the following strings:

- any valid python identifier
- any number of spaces
- (optional) an equals sign followed by any number of spaces and ending with one of the following: any real number, a single quote followed by any number of non-single-quote characters followed by a single quote, or any valid python identifier

Define a function that compiles and returns a regular expression pattern object that matches any valid Python parameter definition.

(Hint: Use the `\w` character class shortcut to keep your regular expression clean.)

To help in debugging, the following examples may be useful. These test cases are a good start, but are not exhaustive. The first table should match valid Python identifiers. The second should match a valid python parameter definition, as defined in this problem. Note that some strings which would be valid in python will not be for this problem.

Matches:	"Mouse"	"compile"	"_123456789"	"_x_"	"while"
Non-matches:	"3rats"	"err*r"	"sq(x)"	"sleep()"	" x"
Matches:	"max=4.2"	"string= '"	"num_guesses"		
Non-matches:	"300"	"is_4=(value==4)"	"pattern = r'^one two fish\$'"		

## Manipulating Text with Regular Expressions

So far we have been solely concerned with whether or not a regular expression and a string match, but the power of regular expressions comes with what can be done with a match. In addition to the `search()` method, regular expression pattern objects have the following useful methods.

Method	Description
<code>match()</code>	Match a regular expression pattern to the beginning of a string.
<code>fullmatch()</code>	Match a regular expression pattern to all of a string.
<code>search()</code>	Search a string for the presence of a pattern.
<code>sub()</code>	Substitute occurrences of a pattern found in a string.
<code>subn()</code>	Same as <code>sub</code> , but also return the number of substitutions made.
<code>split()</code>	Split a string by the occurrences of a pattern.
<code>findall()</code>	Find all occurrences of a pattern in a string.
<code>finditer()</code>	Return an iterator yielding a match object for each match.

Table 3.4: Methods of regular expression pattern objects.

Some substitutions require remembering part of the text that the regular expression matches. Groups are useful here: each group in the regular expression can be represented in the substitution string by `\n`, where `n` is an integer (starting at 1) specifying which group to use.

```
# Find words that start with 'cat', remembering what comes after the 'cat'.
>>> pig_latin = re.compile(r"\bcat(\w*)")
>>> target = "Let's catch some catfish for the cat"

>>> pig_latin.sub(r"at\1clay", target) # \1 = (\w*) from the expression.
"Let's atchclay some atfishclay for the atclay"
```

The repetition operators `?`, `+`, `*`, and `{m,n}` are *greedy*, meaning that they match the largest string possible. On the other hand, the operators `??`, `+`, `*?`, and `{m,n}?` are *non-greedy*, meaning they match the smallest strings possible. This is very often the desired behavior for a regular expression.

```
>>> target = "<abc> <def> <ghi>"

# Match angle brackets and anything in between.
>>> greedy = re.compile(r"<.*>$") # Greedy *
>>> greedy.findall(target)
['<abc> <def> <ghi>'] # The entire string matched!

# Try again, using the non-greedy version.
>>> nongreedy = re.compile(r"<.*?>") # Non-greedy *?
>>> nongreedy.findall(target)
['<abc>', '<def>', '<ghi>'] # Each <> set is an individual match.
```

Finally, there are a few customizations that make searching larger texts manageable. Each of these *flags* can be used as keyword arguments to `re.compile()`.

Flag	Description
<code>re.DOTALL</code>	<code>.</code> matches any character at all, including the newline.
<code>re.IGNORECASE</code>	Perform case-insensitive matching.
<code>re.MULTILINE</code>	<code>^</code> matches the beginning of lines (after a newline) as well as the string; <code>\$</code> matches the end of lines (before a newline) as well as the end of the string.

Table 3.5: Regular expression flags.

A benefit of using '^' and '\$' is that they allow you to search across multiple lines. For example, how would we match "World" in the string "Hello\nWorld"? Using `re.MULTILINE` in the `re.search` function will allow us to match at the beginning of each new line, instead of just the beginning of the string. The following shows how to implement multiline searching:

```
>>>pattern1 = re.compile("^W")
>>>pattern2 = re.compile("^W", re.MULTILINE)
>>>bool(pattern1.search("Hello\nWorld"))
False
>>>bool(pattern2.search("Hello\nWorld"))
True
```

**Problem 5.** A Python *block* is composed of several lines of code with the same indentation level. Blocks are delimited by key words and expressions, followed by a colon. Possible key words are `if`, `elif`, `else`, `for`, `while`, `try`, `except`, `finally`, `with`, `def`, and `class`. Some of these keywords require an expression to precede the colon (`if`, `elif`, `for`, etc. Some require no expressions to precede the colon (`else`, `finally`), and `except` may or may not have an expression before the colon.

Write a function that accepts a string of Python code and uses regular expressions to place colons in the appropriate spots. Assume that every colon is missing in the input string.

```
"""
k, i, p = 999, 1, 0
while k > i
    i *= 2
    p += 1
    if k != 999
        print("k should not have changed")
    else
        pass
print(p)
"""

# The string given above should become this string.
"""
k, i, p = 999, 1, 0
while k > i:
    i *= 2
    p += 1
    if k != 999:
        print("k should not have changed")
    else:
        pass
print(p)
"""
```

**Problem 6.** The file `fake_contacts.txt` contains poorly formatted contact data for 2000 fictitious individuals. Each line of the file contains data for one person, including their name and possibly their birthday, email address, and/or phone number. The formatting of the data is not consistent, and much of it is missing. For example, not everyone has their birthday listed, and those who do may have it listed in the form 1/1/11, 1/01/2011, or some other format.

Use regular expressions to parse the data and format it uniformly, writing birthdays as `mm/dd/yyyy` and phone numbers as `(xxx)xxx-xxxx`. Return a dictionary where the key is the name of an individual and the value is another dictionary containing their information. Each of these inner dictionaries should have the keys `"birthday"`, `"email"`, and `"phone"`. In the case of missing data, map the key to `None`. The first two entries of the completed dictionary are given below.

```
{
    "John Doe": {
        "birthday": "01/01/1990",
        "email": "john_doe90@hopefullynotarealaddress.com",
        "phone": "(123)456-7890"
    },
    "Jane Smith": {
        "birthday": None,
        "email": None,
        "phone": "(222)111-3333"
    },
    # ...
}
```

## Additional Material

### Regular Expressions in the Unix Shell

As we have seen,, regular expressions are very useful when we want to match patterns. Regular expressions can be used when matching patterns in the Unix Shell. Though there are many Unix commands that take advantage of regular expressions, we will focus on **grep** and **awk**.

#### Regular Expressions and grep

Recall from Lab 1 that **grep** is used to match patterns in files or output. It turns out we can use regular expressions to define the pattern we wish to match.

In general, we use the following syntax:

```
$ grep 'regex' filename
```

We can also use regular expressions when piping output to **grep**.

```
# List details of directories within current directory.  
$ ls -l | grep ^d
```

#### Regular Expressions and awk

By incorporating regular expressions, the **awk** command becomes much more robust. Before GUI spreadsheet programs like Microsoft Excel, **awk** was commonly used to visualize and query data from a file.

Including **if** statements inside **awk** commands gives us the ability to perform actions on lines that match a given pattern. The following example prints the filenames of all files that are owned by **freddy**.

```
$ ls -l | awk ' {if ($3 ~ /freddy/) print $9} '
```

Because there is a lot going on in this command, we will break it down piece-by-piece. The output of **ls -l** is getting piped to **awk**. Then we have an **if** statement. The syntax here means if the condition inside the parenthesis holds, print field 9 (the field with the filename). The condition is where we use regular expressions. The **~** checks to see if the contents of field 3 (the field with the username) matches the regular expression found inside the forward slashes. To clarify, **freddy** is the regular expression in this example and the expression must be surrounded by forward slashes.

Consider a similar example. In this example, we will list the names of the directories inside the current directory. (This replicates the behavior of the Unix command **ls -d \*/**)

```
$ ls -l | awk ' {if ($1 ~ /^d/) print $9} '
```

Notice in this example, we printed the names of the directories, whereas in one of the example using **grep**, we printed all the details of the directories as well.



**ACHTUNG!**

Some of the definitions for character classes we used earlier in this lab will not work in the Unix Shell. For example, `\w` and `\d` are not defined. Instead of `\w`, use `[[[:alnum:]]]`. Instead of `\d`, use `[[[:digit:]]]`. For a complete list of similar character classes, search the internet for *POSIX Character Classes* or *Bracket Character Classes*.



# 4

## Web Technologies

**Lab Objective:** *The Internet is a term for the collective grouping of all publicly accessible computer networks in the world. This network can be traversed to access services such as social communication, maps, video streaming, and large datasets, all of which are hosted on computers across the world. Using these technologies requires an understanding of data serialization, data transportation protocols, and how programs such as servers, clients, and APIs are created to facilitate this communication.*

### Data Serialization

*Serialization* is the process of packaging data in a form that makes it easy to transmit the data and quickly reconstruct it on another computer or in a different programming language. Many serialization metalanguages exist, such as Python's `pickle`, YAML, XML, and JSON. JSON, which stands for *JavaScript Object Notation*, is the dominant format for serialization in web applications. Despite having “JavaScript” in its name, JSON is a language-independent format and is frequently used for transmitting data between different programming languages. It stores information about objects as a specially formatted string that is easy for both humans and machines to read and write. *Deserialization* is the process of reconstructing an object from the string.

JSON is built on two types of data structures: a collection of key/value pairs similar to Python's built-in `dict`, and an ordered list of values similar to Python's built-in `list`.

```
{
    "lastname": "Smith",
    "children": [
        {
            "name": "Timmy",
            "age": 8
        },
        {
            "name": "Missy",
            "age": 5
        }
    ]
}
```

# A family's info written in JSON format.  
# The outer dictionary has two keys:  
# "lastname" and "children".  
# The "children" key maps to a list of  
# two dictionaries, one for each of the  
# two children.

## NOTE

To see a longer example of what JSON looks like, try opening a Jupyter Notebook (a `.ipynb` file) in a plain text editor. The file lists the Notebook cells, each of which has attributes like `"cell_type"` (usually code or markdown) and `"source"` (the actual code in the cell).

The JSON libraries of various languages have a fairly standard interface. The Python standard library module for JSON is called `json`. If performance speed is critical, consider using the `ujson` or `simplejson` modules that are written in C. A string written in JSON format that represents a piece of data is called a *JSON message*. The `json.dumps()` function generates the JSON message for a single Python object, which can be stored and used within the Python program. Alternatively, the json encoder `json.dump()` generates the same object, but writes it directly to a file. To load a JSON string or file, use the json decoder `json.loads()` or `json.load()`, respectively.

```
>>> import json

# Store info about a car in a nested dictionary.
>>> my_car = {
...     "car": {
...         "make": "Ford",
...         "color": [255, 30, 30] },
...     "owner": "me" }

# Get the JSON message corresponding to my_car.
>>> car_str = json.dumps(my_car)
>>> car_str
'{"car": {"make": "Ford", "color": [255, 30, 30]}, "owner": "me"}'

# Load the JSON message into a Python object, reconstructing my_car.
>>> car_object = json.loads(car_str)
>>> for key in car_object:          # The loaded object is a dictionary.
...     print(key + ':', car_object[key])
...
car: {'make': 'Ford', 'color': [255, 30, 30]}
owner: me

# Write the car info to an external file.
>>> with open("my_car.json", 'w') as outfile:
...     json.dump(my_car, outfile)
...

# Read the file to check that it saved correctly.
>>> with open("my_car.json", 'r') as infile:
...     new_car = json.load(infile)
...
>>> print(new_car.keys())          # This loaded object is also a dictionary.
dict_keys(['car', 'owner'])
```

**Problem 1.** The file `nyc_traffic.json` contains information about 1000 traffic accidents in New York City during the summer of 2017.<sup>a</sup> Each entry lists one or more reasons for the accident, such as “Unsafe Speed” or “Fell Asleep.”

Write a function that loads the data from the JSON file. Make a readable, sorted bar chart showing the total number of times that each of the 7 most common reasons for accidents are listed in the data set.

(Hint: the `collections.Counter` data structure and use `plt.tight_layout()` may be useful here.)

To check your work, the 6th most common reason is “Backing Unsafely,” listed 59 times.

<sup>a</sup>See <https://opendata.cityofnewyork.us/>.

## Custom Encoders and Decoders for JSON

The default JSON encoder and decoder do not support serialization for every kind of data structure. For example, a `set` cannot be serialized using only `json` functions. However, the default JSON encoder can be subclassed to handle sets or custom data structures. A custom encoder must organize the information in an object as nested lists and dictionaries. The corresponding custom decoder uses the way that the encoder organizes the information to reconstruct the original object.

For example, one way to serialize a `set` is to express it as a dictionary with one key that indicates its data type, and another key mapping to the actual data.

```
>>> class SetEncoder(json.JSONEncoder):
...     """A custom JSON encoder for Python sets."""
...     def default(self, obj):
...         if not isinstance(obj, set):
...             raise TypeError("expected a set for encoding")
...         return {"dtype": "set", "data": list(obj)}
...
# Use the custom encoder to convert a set to its custom JSON message.
>>> set_message = json.dumps(set('abca'), cls=SetEncoder)
>>> set_message
'{"dtype": "set", "data": ["a", "b", "c"]}'

# Define a custom decoder for JSON messages generated by the SetEncoder.
>>> def set_decoder(item):
...     if "dtype" in item:
...         if item["dtype"] != "set" or "data" not in item:
...             raise ValueError("expected a JSON message from SetEncoder")
...         return set(item["data"])
...     raise ValueError("expected a JSON message from SetEncoder")
...
# Use the custom decoder to convert a JSON message to the original object.
>>> json.loads(set_message, object_hook=set_decoder)
{'a', 'b', 'c'}
```

It is good practice to check for errors to ensure that custom encoders and decoders are only used when intended.

**Problem 2.** The following class facilitates a regular  $3 \times 3$  game of tic-tac-toe, where the boxes in the board have the following coordinates.

(0,0)	(0,1)	(0,2)
(1,0)	(1,1)	(1,2)
(2,0)	(2,1)	(2,2)

Write a custom encoder and decoder for the `TicTacToe` class. If the custom encoder receives anything other than a `TicTacToe` object, raise a `TypeError`.

```
class TicTacToe:
    def __init__(self):
        """Initialize an empty board. The 0's go first."""
        self.board = [[' ']*3 for _ in range(3)]
        self.turn, self.winner = "0", None

    def move(self, i, j):
        """Mark an 0 or X in the (i,j)th box and check for a winner."""
        if self.winner is not None:
            raise ValueError("the game is over!")
        elif self.board[i][j] != ' ':
            raise ValueError("space ({},{}) already taken".format(i,j))
        self.board[i][j] = self.turn

        # Determine if the game is over.
        b = self.board
        if any(sum(s == self.turn for s in r)==3 for r in b):
            self.winner = self.turn # 3 in a row.
        elif any(sum(r[i] == self.turn for r in b)==3 for i in range(3)):
            self.winner = self.turn # 3 in a column.
        elif b[0][0] == b[1][1] == b[2][2] == self.turn:
            self.winner = self.turn # 3 in a diagonal.
        elif b[0][2] == b[1][1] == b[2][0] == self.turn:
            self.winner = self.turn # 3 in a diagonal.
        else:
            self.turn = "0" if self.turn == "X" else "X"

    def empty_spaces(self):
        """Return the list of coordinates for the empty boxes."""
        return [(i,j) for i in range(3) for j in range(3)
                if self.board[i][j] == ' ']

    def __str__(self):
        return "\n-----\n".join(" | ".join(r) for r in self.board)
```

## Servers and Clients

The Internet has specific protocols that allow for standardized communication within and between computers. The most common communication protocols in computer networks are contained in the Internet Protocol Suite. Among these is *Transmission Control Protocol* (TCP), which is used to establish a connection between two computers, exchange bits of information called *packets*, and then close the connection. TCP creates the connection via network *socket* objects that are used to send and receive data packets from a computer.

Essentially, this can be thought of as a PO box at a post office. The socket is like a PO box owned by a particular program, which checks it periodically for updates. The computer can be thought of as the post office which houses the PO boxes. PO boxes, or sockets, can send mail to each other within the same post office, or computer, easily, but more work is needed when the PO boxes send mail to each other from one post office, or computer, to another.

A *server* is a program that interacts with and provides functionality to client programs. This can be thought of as the PO box which sends the mail. A *client* program contacts a server to receive some sort of response that assists it in fulfilling its function. This can be thought of as the PO box which receives the mail. Unlike with physical mail, in which the sender can send mail to himself, a socket being used as a server in a computer cannot also serve as a client at the same time. Servers are fundamental to modern networks and provide services such as file sharing, authentication, webpage information, databases, etc.

## Creating a Server

One simple way to create a server in Python is via the `socket` module. The server socket must first be initialized by specifying the type of connection and the address at which clients can find the server. The server socket then listens and waits for a connection from a client, receives and processes data, and eventually sends a response back to the client. After exchanges between the server and the client are finished, the server closes the connection to the client.

Name	Description
<code>socket</code>	Create a new socket using the given address family, socket type and protocol number.
<code>bind</code>	Bind the socket to an address. The socket must not already be bound.
<code>listen</code>	Enable a server to accept connections.
<code>accept</code>	Accept a connection. Must be bound to an address and listening for connections.
<code>connect</code>	Connect to a remote socket at address.
<code>sendall</code>	Send data to the socket. The socket must be connected to a remote socket.
	Continues to send data until either all data has been sent or an error occurs.
<code>recv</code>	Receive data from the socket. Must be given a buffer size; use 1024.
<code>close</code>	Mark the socket closed.

Table 4.1: Socket method descriptions

The `socket.socket()` method receives two parameters, which specify the socket type. The server address is a (host, port) tuple. The host is the IP address, which in this case is `"localhost"` or `"0.0.0.0"`—the default address that specifies the local machine and allows connections on all interfaces. The port number is an integer from 0 to 65535. About 250 port numbers are commonly used, and certain ports have pre-defined uses. Only use port numbers greater than 1023 to avoid interrupting standard system services, such as email and system updates.

After setting up the server socket, the server program waits for a client to connect. The `accept()` method returns a new socket object and the client's address. Data is received through the connection socket's `recv()` method, which takes an integer specifying the number of bits of data to receive. The data is transferred as a raw byte stream (of type `bytes`), so the `decode()` method is necessary to translate the data into a string. Likewise, data that is sent back to the client through the connection socket's `sendall()` method must be encoded into a byte stream via the `encode()` method.

Finally, `try-finally` blocks in the server ensure that the connection is always closed securely. Put these blocks within an infinite `while(True)` block to ensure that your server will be ready for any client request. Note that the `accept()` method does not return until a connection is made with a client. Therefore, this server program cannot be executed in its entirety without a client. To stop a server, raise a `KeyboardInterrupt` (press `ctrl+c`) in the terminal where it is running.

Note that server-client communication is the reason that JSON serialization and deserialization is so important. For example, information such as an image or a family tree could be sent more simply using serialized objects.

```
def mirror_server(server_address=("0.0.0.0", 33333)):
    """A server for reflecting strings back to clients in reverse order."""
    print("Starting mirror server on {}".format(server_address))

    # Specify the socket type, which determines how clients will connect.
    server_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_sock.bind(server_address)      # Assign this socket to an address.
    server_sock.listen(1)                 # Start listening for clients.

    while True:
        # Wait for a client to connect to the server.
        print("\nWaiting for a connection...")
        connection, client_address = server_sock.accept()
        try:
            # Receive data from the client.
            print("Connection accepted from {}".format(client_address))
            in_data = connection.recv(1024).decode()      # Receive data.
            print("Received '{}' from client".format(in_data))

            # Process the received data and send something back to the client.
            out_data = in_data[::-1]
            print("Sending '{}' back to the client".format(out_data))
            connection.sendall(out_data.encode())          # Send data.

        finally:      # Make sure the connection is closed securely.
            connection.close()
            print("Closing connection from {}".format(client_address))
```



**ACHTUNG!**

It often takes some time for a computer to reopen a port after closing a server connection. This is due to the timeout functionality of specific protocols that check connections for errors and disruptions. While testing code, wait a few seconds before running the program again, or use different ports for each test.

**Problem 3.** Write a function that accepts a (host, port) tuple and starts up a tic-tac-toe server at the specified location. Wait to accept a connection, then while the connection is open, repeat the following operations.

1. Receive a JSON serialized `TicTacToe` object (serialized with your custom encoder from Problem 2) from the client.
2. Deserialize the `TicTacToe` object using your custom decoder from Problem 2.
3. If the client has just won the game, send **"WIN"** back to the client and close the connection.
4. If there is no winner but board is full, send **"DRAW"** to the client and close the connection.
5. If the game still isn't over, make a random move on the tic-tac-toe board and serialize the updated `TicTacToe` object. If this move wins the game, send **"LOSE"** to the client, then send the serialized object separately (as proof), and close the connection. Otherwise, send only the updated `TicTacToe` object back to the client but keep the connection open.

(Hint: print information at each step so you can see what the server is doing.)

Ensure that the connection closes securely even if an exception is raised. Note that you will not be able to fully test your server until you have written a client (see Problem 4).

## Creating a Client

The `socket` module also has tools for writing client programs. First, create a socket object with the same settings as the server socket, then call the `connect()` method with the server address as a parameter. Once the client socket is connected to the server socket, the two sockets can transfer information between themselves.

Unlike the server socket, the client socket sends and reads the data itself instead of creating a new connection socket. When the client program is complete, close the client socket. The server will keep running, waiting for another client to serve.

To see a client and server communicate, open a terminal and run the server. Then run the client in a separate terminal. Try this with the provided examples.

```
def mirror_client(server_address=("0.0.0.0", 33333)):
    """A client program for mirror_server()."""
    print("Attempting to connect to server at {}".format(server_address))

    # Set up the socket to be the same type as the server.
```

```

client_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_sock.connect(server_address)      # Attempt to connect to the server.

# Send some data from the client user to the server.
out_data = input("Type a message to send to the server: ")
client_sock.sendall(out_data.encode())   # Send data.

# Wait to receive a response back from the server.
in_data = client_sock.recv(1024).decode() # Receive data.
print("Received '{}' from the server".format(in_data))

# Close the client socket.
client_sock.close()

```

**Problem 4.** Write a client function that accepts a (host, port) tuple and connects to the tic-tac-toe server at the specified location. Start by initializing a new TicTacToe object, then repeat the following steps until the game is over.

1. Print the board and prompt the player for a move. Continue prompting the player until they provide valid input.
2. Update the board with the player's move, then serialize it using your custom encoder from Problem 2, and send the serialized version to the server.
3. Receive a response from the server. If the game is over, congratulate or mock the player appropriately. If the player lost, receive a second response from the server (the final game board), deserialize it, and print it out.

Close the connection once the game ends.

## APIs

An *Application Program Interface* (API) is a particular kind of server that listens for requests from authorized users and responds with data. For example, a list of locations can be sent with the proper request syntax to a Google Maps API, and it will respond with the calculated driving time from start to end, including each location. Every API has *endpoints* where clients send their requests. Though standards exist for creating and communicating with APIs, most APIs have a unique syntax for authentication and requests that is documented by the organization providing the service.

The `requests` module is the standard way to send a download request to an API in Python.

```

>>> import requests
>>> requests.get(endpoint).json()    # Download and extract the data.

```

**ACHTUNG!**

Each website and API has a policy that specifies appropriate behavior for automated data retrieval and usage. If data is requested without complying with these requirements, there can be severe legal consequences. Most websites detail their policies in a file called *robots.txt* on their main page. See, for example, <https://www.google.com/robots.txt>.

## Additional Material

### Other Internet Protocols

There are many protocols in the Internet Protocol Suite other than TCP that are used for different purposes. The Protocol Suite can be divided into four categorical layers:

1. **Application:** Software that utilizes transport protocols to move information between computers. This layer includes protocols important for email, file transfers, and browsing the web.
2. **Transport:** Protocols that assist in basic high level communication between two computers in areas such as data-streaming, reliability control, and flow control.
3. **Internet:** Protocols that handle routing, assignment of addresses, and movement of data on a network.
4. **Link:** Protocols that communicate with local networking hardware such as routers and switches.

Although these examples are simple, every data transfer with TCP follows a similar pattern. For basic connections, these interactions are simple processes. However, requesting a webpage would require management of possibly hundreds of connections. In order to make this more feasible, there are higher level protocols that handle smaller TCP/IP details. The most predominant of these protocols is HTTP.

## HTTP

HTTP stands for Hypertext Transfer Protocol, which is an application layer networking protocol. It is a higher level protocol than TCP but uses TCP protocols to manage connections and provide network capabilities. The protocol is centered around a request and response paradigm in which a client makes a request to a server and the server replies with response. There are several methods, or *requests*, defined for HTTP servers, the three most common of which are GET, POST, and PUT. GET requests request information from a server, POST requests modify the state of the server, and PUT requests add new pieces of data to the server.

Every HTTP request or response consists of two parts: a header and a body. The headers contain important information about the request including: the type of request, encoding, and a timestamp. Custom headers may be added to any request to provide additional information. The body of the request or response contains the appropriate data or may be empty.

An HTTP connection can be setup in Python by using the standard Python library `http`. Though it is the standard, the process can be greatly simplified by using an additional library called `requests`. The following demonstrates a simple GET request with the `http` library.

```
>>> import http
>>> conn = http.client.HTTPConnection("www.example.net") # Establish connection
>>> conn.request("GET", "/") # Send GET request
>>> resp = conn.getresponse() # Server response message
>>> print(resp.status)
200 # A status of 200 is the standard sign for successful communication
>>> print(resp.getheaders())
[('Cache-Control', 'max-age=604800'), ... , ('Content-Length', '1270')] # ←
    Header information about request
>>> print(resp.read())
```

```
b'<!doctype html>\n<html> ... n</html>\n'      # Long string with HTML from ↵
webpage
>>> conn.close() # When the request is finished, the connection is closed
```

As previously mentioned, this exchange is greatly simplified by the `requests` library:

```
>>> import requests
>>> r = requests.get("http://www.example.net")
>>> print(r.headers)
{'Cache-Control': 'max-age=604800', ... , 'Content-Length': '606'}
>>> print(r.content)
b'<!doctype html>\n<html> ... n</html>\n'
```

This process is how a web browser (a client program) retrieves a webpage. It first sends an HTTP request to the web server (a server program) and receives the HTML, CSS, and other code files for a webpage, which are compiled and run in the web browser.

Requests also often include parameters which are keys to tell the server what is being requested or placed. These parameters can be included in the URL that requests from the server, or in parameters that the `requests` library can implement. For example:

```
>>> r = requests.get("http://httpbin.org/get?key2=value2&key1=value1")
>>> print(r.text)
{
  "args": {
    "key1": "value1",
    "key2": "value2"
  },
  ...
},
  "origin": "128.187.116.7",
  "url": "http://httpbin.org/get?key2=value2&key1=value1"
}
>>> r = requests.get("http://httpbin.org/get", params={'key1': 'value1', 'key2': '↵
value2'})
>>> print(r.url)
http://httpbin.org/get?key2=value2&key1=value1
>>> print(r.text)
{
  "args": {
    "key1": "value1",
    "key2": "value2"
  },
  ...
},
  "origin": "128.187.116.7",
  "url": "http://httpbin.org/get?key2=value2&key1=value1"
}
```

A similar format to GET requests can also be used for PUT or POST requests. These special requests alter the state of the server or send a piece of data to the server, respectively. In addition, for PUT and POST requests, a data string or dictionary may be sent as a binary stream attachment. The `requests` library attaches these data objects with the `data` parameter. For example:

```
>>> r = requests.put('http://httpbin.org/put', data='{key1:value1,key2:value2}' ↵
)
>>> print(r.text)
{
  "args": {},
  "data": "{key1:value1,key2:value2}",
  "files": {},
  "form": {},
  ...
  "json": null,
  "origin": "128.187.116.7",
  "url": "http://httpbin.org/put"
}
```

Note that the `data` parameter accepts input in the form of a JSON string.

Frequently, when these requests arrive at the server, they are in the form of a binary stream, which can be read with similar notation to the Python `open` function. Below is an example of reading the previous PUT request with a data attachment as a binary stream using `read`.

```
>>> data = r.json()['data'] # Retrieve the sent data string
>>> print(data)
'{key1:value1,key2:value2}'
>>> print(len(data.encode())) # Show the string's length in bytes
25
>>> with open('request.txt', 'w') as file:
>>>     file.write(data) # Write the string to a file
>>> with open('request.txt', 'rb') as file: # Open the file as a binary stream
>>>     file.read(25) # Read the correct number of bytes
b'{key1:value1,key2:value2}'
```

For more information on the `requests` library, see the documentation at <http://docs.python-requests.org/>.

# 5

## Web Scraping

**Lab Objective:** *Web Scraping is the process of gathering data from websites on the internet. Since almost everything rendered by an internet browser as a web page uses HTML, the first step in web scraping is being able to extract information from HTML. In this lab, we introduce the requests library for scraping web pages, and BeautifulSoup, Python's canonical tool for efficiently and cleanly navigating and parsing HTML.*

### HTTP and Requests

HTTP stands for Hypertext Transfer Protocol, which is an application layer networking protocol. It is a higher level protocol than TCP, which we used to build a server in the Web Technologies lab, but uses TCP protocols to manage connections and provide network capabilities. The HTTP protocol is centered around a request and response paradigm, in which a client makes a request to a server and the server replies with a response. There are several methods, or *requests*, defined for HTTP servers, the three most common of which are GET, POST, and PUT. GET requests request information from the server, POST requests modify the state of the server, and PUT requests add new pieces of data to the server.

The standard way to get the source code of a website using Python is via the `requests` library.<sup>1</sup> Calling `requests.get()` sends an HTTP GET request to a specified website. The website returns a response code, which indicates whether or not the request was received, understood, and accepted. If the response code is good, typically 200, then the response will also include the website source code as an HTML file.

```
>>> import requests

# Make a request and check the result. A status code of 200 is good.
>>> response = requests.get("http://www.byu.edu")
>>> print(response.status_code, response.ok, response.reason)
200 True OK
```

<sup>1</sup>Though `requests` is not part of the standard library, it is recognized as a standard tool in the data science community. See <http://docs.python-requests.org/>.

```
# The HTML of the website is stored in the 'text' attribute.
>>> print(response.text)
<!DOCTYPE html>
<html lang="en" dir="ltr" prefix="content: http://purl.org/rss/1.0/modules/↵
content/ dc: http://purl.org/dc/terms/ foaf: http://xmlns.com/foaf/0.1/ ↵
og: http://ogp.me/ns# rdfs: http://www.w3.org/2000/01/rdf-schema# schema:↵
http://schema.org/ sioc: http://rdfs.org/sioc/ns# sioc: http://rdfs.org↵
/sioc/types# skos: http://www.w3.org/2004/02/skos/core# xsd: http://www.↵
w3.org/2001/XMLSchema# " class=" ">

<head>
  <meta charset="utf-8" />
# ...
```

Note that some websites aren't built to handle large amounts of traffic or many repeated requests. Most are built to identify web scrapers or crawlers that initiate many consecutive GET requests without pauses, and retaliate or block them. When web scraping, always make sure to store the data that you receive in a file and include error checks to prevent retrieving the same data unnecessarily. This is especially important in larger applications.

**Problem 1.** Use the `requests` library to get the HTML source for the website `http://www.example.com`. Save the source as a file called `example.html`. If the file already exists, make sure not to scrape the website, or overwrite the file. You will use this file later in the lab.

## ACHTUNG!

Scraping copyrighted information without the consent of the copyright owner can have severe legal consequences. Many websites, in their terms and conditions, prohibit scraping parts or all of the site. Websites that do allow scraping usually have a file called `robots.txt` (for example, `www.google.com/robots.txt`) that specifies which parts of the website are off-limits and how often requests can be made according to the *robots exclusion standard*.<sup>a</sup>

Be careful and considerate when doing any sort of scraping, and take care when writing and testing code to avoid unintended behavior. It is up to the programmer to create a scraper that respects the rules found in the terms and conditions and in `robots.txt`.<sup>b</sup>

<sup>a</sup>See [www.robotstxt.org/orig.html](http://www.robotstxt.org/orig.html) and [en.wikipedia.org/wiki/Robots\\_exclusion\\_standard](http://en.wikipedia.org/wiki/Robots_exclusion_standard).

<sup>b</sup>Python provides a parsing library called `urllib.robotparser` for reading `robot.txt` files. For more information, see <https://docs.python.org/3/library/urllib.robotparser.html>.



## HTML

*Hyper Text Markup Language*, or *HTML*, is the standard *markup language*—a language designed for the processing, definition, and presentation of text—for creating webpages. It structures a document using pairs of *tags* that surround and define content. Opening tags have a tag name surrounded by angle brackets (`<tag-name>`). The companion closing tag looks the same, but with a forward slash before the tag name (`</tag-name>`). A list of all current HTML tags can be found at <http://htmldog.com/reference/htmltags>.

Most tags can be combined with *attributes* to include more data about the content, help identify individual tags, and make navigating the document much simpler. In the following example, the `<a>` tag has `id` and `href` attributes.

```
<html>                                <!-- Opening tags -->
  <body>
    <p>
      Click <a id='info' href='http://www.example.com'>here</a>
      for more information.
    </p>                                <!-- Closing tags -->
  </body>
</html>
```

In HTML, `href` stands for *hypertext reference*, a link to another website. Thus the above example would be rendered by a browser as a single line of text, with **here** being a clickable link to <http://www.example.com>:

Click here for more information.

Unlike Python, HTML does not enforce indentation (or any whitespace rules), though indentation generally makes HTML more readable. The previous example can be written in a single line.

```
<html><body><p>Click <a id='info' href='http://www.example.com/info'>here</a>
for more information.</p></body></html>
```

Special tags, which don't contain any text or other tags, are written without a closing tag and in a single pair of brackets. A forward slash is included between the name and the closing bracket. Examples of these include `<hr/>`, which describes a horizontal line, and `<img/>`, the tag for representing an image.

**Problem 2.** Using the output from Problem 1, examine the HTML source code for <http://www.example.com>. What tags are used? What is the value of the `type` attribute associated with the `style` tag?

Write a function that returns the set of names of tags used in the website, and the value of the `type` attribute of the `style` tag (as a string).  
(Hint: there are ten unique tag names.)

## BeautifulSoup

BeautifulSoup (bs4) is a package<sup>2</sup> that makes it simple to navigate and extract data from HTML documents. See <http://www.crummy.com/software/BeautifulSoup/bs4/doc/index.html> for the full documentation.

The `bs4.BeautifulSoup` class accepts two parameters to its constructor: a string of HTML code and an HTML parser to use under the hood. The HTML parser is technically a keyword argument, but the constructor prints a warning if one is not specified. The standard choice for the parser is `"html.parser"`, which means the object uses the standard library's `html.parser` module as the engine behind the scenes.

### NOTE

Depending on project demands, a parser other than `"html.parser"` may be useful. A couple of other options are `"lxml"`, an extremely fast parser written in C, and `"html5lib"`, a slower parser that treats HTML in much the same way a web browser does, allowing for irregularities. Both must be installed independently; see <https://www.crummy.com/software/BeautifulSoup/bs4/doc/#installing-a-parser> for more information.

A BeautifulSoup object represents an HTML document as a tree. In the tree, each tag is a *node* with nested tags and strings as its *children*. The `prettify()` method returns a string that can be printed to represent the BeautifulSoup object in a readable format that reflects the tree structure.

```
>>> from bs4 import BeautifulSoup

>>> small_example_html = """
<html><body><p>
    Click <a id='info' href='http://www.example.com'>here</a>
    for more information.
</p></body></html>
"""

>>> small_soup = BeautifulSoup(small_example_html, 'html.parser')
>>> print(small_soup.prettify())
<html>
<body>
<p>
    Click
    <a href="http://www.example.com" id="info">
        here
    </a>
    for more information.
</p>
</body>
</html>
```

<sup>2</sup>BeautifulSoup is not part of the standard library; install it with `conda install beautifulsoup4` or with `pip install beautifulsoup4`.

Each tag in a BeautifulSoup object's HTML code is stored as a `bs4.element.Tag` object, with actual text stored as a `bs4.element.NavigableString` object. Tags are accessible directly through the BeautifulSoup object.

```
# Get the <p> tag (and everything inside of it).
>>> small_soup.p
<p>
    Click <a href="http://www.example.com" id="info">here</a>
    for more information.
</p>

# Get the <a> sub-tag of the <p> tag.
>>> a_tag = small_soup.p.a
>>> print(a_tag, type(a_tag), sep='\n')
<a href="http://www.example.com" id="info">here</a>
<class 'bs4.element.Tag'>

# Get just the name, attributes, and text of the <a> tag.
>>> print(a_tag.name, a_tag.attrs, a_tag.string, sep="\n")
a
{'id': 'info', 'href': 'http://www.example.com'}
here
```

Attribute	Description
<code>name</code>	The name of the tag
<code>attrs</code>	A dictionary of the attributes
<code>string</code>	The single string contained in the tag
<code>strings</code>	Generator for strings of children tags
<code>stripped_strings</code>	Generator for strings of children tags, stripping whitespace
<code>text</code>	Concatenation of strings from all children tags

Table 5.1: Data attributes of the `bs4.element.Tag` class.

**Problem 3.** The BeautifulSoup class has a `find_all()` method that, when called with `True` as the only argument, returns a list of all tags in the HTML source code.

Write a function that accepts a string of HTML code as an argument. Use BeautifulSoup to return a list of the **names** of the tags in the code. Use your function and the source code from `http://www.example.com` (use the output from Problem 1) to check your answers from Problem 2.

## Navigating the Tree Structure

Not all tags are easily accessible from a BeautifulSoup object. Consider the following example.

```
>>> pig_html = """
<html><head><title>Three Little Pigs</title></head>
<body>
<p class="title"><b>The Three Little Pigs</b></p>
<p class="story">Once upon a time, there were three little pigs named
<a href="http://example.com/larry" class="pig" id="link1">Larry,</a>
<a href="http://example.com/mo" class="pig" id="link2">Mo</a>, and
<a href="http://example.com/curly" class="pig" id="link3">Curly.</a>
<p>The three pigs had an odd fascination with experimental construction.</p>
<p>...</p>
</body></html>
"""

>>> pig_soup = BeautifulSoup(pig_html, "html.parser")
>>> pig_soup.p
<p class="title"><b>The Three Little Pigs</b></p>

>>> pig_soup.a
<a class="pig" href="http://example.com/larry" id="link1">Larry,</a>
```

Since the HTML in this example has several `<p>` and `<a>` tags, only the **first** tag of each name is accessible directly from `pig_soup`. The other tags can be accessed by manually navigating through the HTML tree.

Every HTML tag (except for the topmost tag, which is usually `<html>`) has a *parent* tag. Each tag also has zero or more *sibling* and *children* tags or text. Following a true tree structure, every `bs4.element.Tag` in a soup has multiple attributes for accessing or iterating through parent, sibling, or child tags.

Attribute	Description
<code>parent</code>	The parent tag
<code>parents</code>	Generator for the parent tags up to the top level
<code>next_sibling</code>	The tag immediately after to the current tag
<code>next_siblings</code>	Generator for sibling tags after the current tag
<code>previous_sibling</code>	The tag immediately before the current tag
<code>previous_siblings</code>	Generator for sibling tags before the current tag
<code>contents</code>	A list of the immediate children tags
<code>children</code>	Generator for immediate children tags
<code>descendants</code>	Generator for all children tags (recursively)

Table 5.2: Navigation attributes of the `bs4.element.Tag` class.

```
# Start at the first <a> tag in the soup.
>>> a_tag = pig_soup.a
>>> a_tag
<a class="pig" href="http://example.com/larry" id="link1">Larry,</a>

# Get the names of all of <a>'s parent tags, traveling up to the top.
```

```
# The name '[document]' means it is the top of the HTML code.
>>> [par.name for par in a_tag.parents]      # <a>'s parent is <p>, whose
['p', 'body', 'html', '[document]']         # parent is <body>, and so on.

# Get the next siblings of <a>.
>>> a_tag.next_sibling
'\n'                                         # The first sibling is just text.
>>> a_tag.next_sibling.next_sibling         # The second sibling is a tag.
<a class="pig" href="http://example.com/mo" id="link2">Mo</a>

%# Alternatively, get all siblings past <a> at once.
%>>> list(a_tag.next_siblings)
%['\n',
% <a class="pig" href="http://example.com/mo" id="link2">Mo</a>,
% ', and\n',
% <a class="pig" href="http://example.com/curly" id="link3">Curly.</a>,
% '\n',
% <p>The three pigs had an odd fascination with experimental construction.</p>,
% '\n',
% <p>...</p>,
% '\n']
```

Note carefully that newline characters are considered to be children of a parent tag. Therefore iterating through children or siblings often requires checking which entries are tags and which are just text.

```
# Get to the <p> tag that has class="story".
>>> p_tag = pig_soup.body.p.next_sibling.next_sibling
>>> p_tag.attrs["class"]                     # Make sure it's the right tag.
['story']

# Iterate through the child tags of <p> and print hrefs whenever they exist.
>>> for child in p_tag.children:
...     if hasattr(child, "href") and "href" in child.attrs:
...         print(child.attrs["href"])
http://example.com/larry
http://example.com/mo
http://example.com/curly
```

Note that the `"class"` attribute of the `<p>` tag is a list. This is because the `"class"` attribute can take on several values at once; for example, the tag `<p class="story book">` is of class `'story'` and of class `'book'`.

The behavior of the `string` attribute of a `bs4.element.Tag` object depends on the structure of the corresponding HTML tag.

1. If the tag has a string of text and no other child elements, then `string` is just that text.
2. If the tag has exactly one child tag and the child tag has only a string of text, then the tag has the same `string` as its child tag.

3. If the tag has more than one child, then `string` is `None`. In this case, use `strings` to iterate through the child strings. Alternatively, the `get_text()` method returns all text belonging to a tag and to all of its descendants. In other words, it returns anything inside a tag that isn't another tag.

```
>>> pig_soup.head
<head><title>Three Little Pigs</title></head>

# Case 1: the <title> tag's only child is a string.
>>> pig_soup.head.title.string
'Three Little Pigs'

# Case 2: The <head> tag's only child is the <title> tag.
>>> pig_soup.head.string
'Three Little Pigs'

# Case 3: the <body> tag has several children.
>>> pig_soup.body.string is None
True
>>> print(pig_soup.body.get_text().strip())
The Three Little Pigs
Once upon a time, there were three little pigs named
Larry,
Mo, and
Curly.
The three pigs had an odd fascination with experimental construction.
...
```

**Problem 4.** Using the output from Problem 1, write a function that reads the file and loads the code into BeautifulSoup. Find the only `<a>` tag with a hyperlink, and return its text.

## Searching for Tags

Navigating the HTML tree manually can be helpful for gathering data out of lists or tables, but these kinds of structures are usually buried deep in the tree. The `find()` and `find_all()` methods of the BeautifulSoup class identify tags that have distinctive characteristics, making it much easier to jump straight to a desired location in the HTML code. The `find()` method only returns the **first** tag that matches a given criteria, while `find_all()` returns a list of all matching tags. Tags can be matched by name, attributes, and/or text.

```
# Find the first <b> tag in the soup.
>>> pig_soup.find(name='b')
<b>The Three Little Pigs</b>

# Find all tags with a class attribute of 'pig'.
# Since 'class' is a Python keyword, use 'class_' as the argument.
```

```
>>> pig_soup.find_all(class_="pig")
[<a class="pig" href="http://example.com/larry" id="link1">Larry,</a>,
  <a class="pig" href="http://example.com/mo" id="link2">Mo</a>,
  <a class="pig" href="http://example.com/curly" id="link3">Curly.</a>]

# Find the first tag that matches several attributes.
>>> pig_soup.find(attrs={"class": "pig", "href": "http://example.com/mo"})
<a class="pig" href="http://example.com/mo" id="link2">Mo</a>

# Find the first tag whose text is 'Mo'.
>>> pig_soup.find(string='Mo')
'Mo'
# The result is the actual string,
>>> pig_soup.find(string='Mo').parent # so go up one level to get the tag.
<a class="pig" href="http://example.com/mo" id="link2">Mo</a>
```

**Problem 5.** The file `san_diego_weather.html` contains the HTML source for an old page from Weather Underground.<sup>a</sup> Write a function that reads the file and loads it into BeautifulSoup. Return a list of the following tags:

1. The tag containing the date “Thursday, January 1, 2015”.
2. The tags which contain the **links** “Previous Day” and “Next Day.”
3. The tag which contains the number associated with the Actual Max Temperature.

This HTML tree is significantly larger than the previous examples. To get started, consider opening the file in a web browser. Find the element that you are searching for on the page, right click it, and select **Inspect**. This opens the HTML source at the element that the mouse clicked on.

<sup>a</sup>See [http://www.wunderground.com/history/airport/KSAN/2015/1/1/DailyHistory.html?req\\_city=San+Diego&req\\_state=CA&req\\_statename=California&reqdb.zip=92101&reqdb.magic=1&reqdb.wmo=99999&MR=1](http://www.wunderground.com/history/airport/KSAN/2015/1/1/DailyHistory.html?req_city=San+Diego&req_state=CA&req_statename=California&reqdb.zip=92101&reqdb.magic=1&reqdb.wmo=99999&MR=1)

## Advanced Search Techniques

Consider the problem of finding the tag that is a link to the URL `http://example.com/curly`.

```
>>> pig_soup.find(href="http://example.com/curly")
<a class="pig" href="http://example.com/curly" id="link3">Curly.</a>
```

This approach works, but it requires entering in the entire URL. To perform generalized searches, the `find()` and `find_all()` method also accept compiled regular expressions from the `re` module. This way, the methods locate tags whose name, attributes, and/or string matches a pattern.

```
>>> import re

# Find the first tag with an href attribute containing 'curly'.
```

```
>>> pig_soup.find(href=re.compile(r"curly"))
<a class="pig" href="http://example.com/curly" id="link3">Curly.</a>

# Find the first tag with a string that starts with 'Cu'.
>>> pig_soup.find(string=re.compile(r"^Cu")).parent
<a class="pig" href="http://example.com/curly" id="link3">Curly.</a>

# Find all tags with text containing 'Three'.
>>> [tag.parent for tag in pig_soup.find_all(string=re.compile(r"Three"))]
[<title>Three Little Pigs</title>, <b>The Three Little Pigs</b>]
```

Finally, to find a tag that has a particular attribute, regardless of the actual value of the attribute, use `True` in place of search values.

```
# Find all tags with an 'id' attribute.
>>> pig_soup.find_all(id=True)
[<a class="pig" href="http://example.com/larry" id="link1">Larry,</a>,
 <a class="pig" href="http://example.com/mo" id="link2">Mo</a>,
 <a class="pig" href="http://example.com/curly" id="link3">Curly.</a>]

# Find the names all tags WITHOUT an 'id' attribute.
>>> [tag.name for tag in pig_soup.find_all(id=False)]
['html', 'head', 'title', 'body', 'p', 'b', 'p', 'p', 'p']
```

**Problem 6.** The file `large_banks_index.html` is an index of data about large banks, as recorded by the Federal Reserve.<sup>a</sup> Write a function that reads the file and loads the source into BeautifulSoup. Return a list of the tags containing the links to bank data from September 30, 2003 to December 31, 2014, where the dates are in reverse chronological order.

<sup>a</sup>See <https://www.federalreserve.gov/releases/lbr/>.

**Problem 7.** The file `large_banks_data.html` is one of the pages from the index in Problem 6.<sup>a</sup> Write a function that reads the file and loads the source into BeautifulSoup. Create a single figure with two subplots:

1. A sorted bar chart of the seven banks with the most domestic branches.
2. A sorted bar chart of the seven banks with the most foreign branches.

In the case of a tie, sort the banks alphabetically by name.

<sup>a</sup>See <http://www.federalreserve.gov/releases/lbr/20030930/default.htm>.



# 6

## Web Crawling

**Lab Objective:** *Gathering data from the internet often requires information from several web pages. In this lab, we present two methods for crawling through multiple web pages without violating copyright laws or straining the load on a server. We also demonstrate how to scrape data from asynchronously loaded web pages and how to interact programmatically with web pages when needed.*

### Scraping Etiquette

There are two main ways that web scraping can be problematic for a website owner.

1. The scraper doesn't respect the website's terms and conditions, or it gathers private or proprietary data.
2. The scraper imposes too much extra server load by making requests too often or in quick succession.

These are extremely important considerations in any web scraping program. Scraping copyrighted information without the consent of the copyright owner can have severe legal consequences. Many websites, in their terms and conditions, prohibit scraping parts or all of the site. Websites that do allow scraping usually have a file called `robots.txt` (for example, [www.google.com/robots.txt](http://www.google.com/robots.txt)) that specifies which parts of the website are off-limits, and how often requests can be made according to the *robots exclusion standard*.<sup>1</sup>

#### ACHTUNG!

Be careful and considerate when doing any sort of scraping, and take care when writing and testing code to avoid unintended behavior. It is up to the programmer to create a scraper that respects the rules found in the terms and conditions and in `robots.txt`. Make sure to scrape websites legally. <sup>a</sup>

<sup>a</sup>Python provides a parsing library called `urllib.robotparser` for reading `robot.txt` files. For more information, see <https://docs.python.org/3/library/urllib.robotparser.html>.

<sup>1</sup>See [www.robotstxt.org/orig.html](http://www.robotstxt.org/orig.html) and [en.wikipedia.org/wiki/Robots\\_exclusion\\_standard](http://en.wikipedia.org/wiki/Robots_exclusion_standard).

The standard way to get the HTML source code of a website using Python is via the `requests` library.<sup>2</sup> Calling `requests.get()` sends an HTTP GET request to a specified website; the result is an object with a response code that indicates whether or not the request was successful and if so, access to the website source code.

```
>>> import requests

# Make a request and check the result. A status code of 200 is good.
>>> response = requests.get("http://www.example.com")
>>> print(response.status_code, response.ok, response.reason)
200 True OK

# The HTML of the website is stored in the 'text' attribute.
>>> print(response.text)
<!doctype html>
<html>
<head>
  <title>Example Domain</title>

  <meta charset="utf-8" />
  <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
# ...
```

Recall that consecutive GET requests without pauses can strain a website's server and provoke retaliation. Most servers are designed to identify such scrapers, block their access, and sometimes even blacklist the user. This is especially common in smaller websites that aren't built to handle enormous amounts of traffic. To briefly pause the program between requests, use `time.sleep()`.

```
>>> import time
>>> time.sleep(3)                # Pause execution for 3 seconds.
```

The amount of necessary wait time depends on the website. Sometimes, `robots.txt` contains a `Request-rate` directive which gives a ratio of the form `<requests>/<seconds>`. If this doesn't exist, pausing for a half-second to a second between requests is typically sufficient. An email to the site's webmaster is always the safest approach and may be necessary for large scraping operations.

## Crawling Through Multiple Pages

While web *scraping* refers to the actual gathering of web-based data, web *crawling* refers to the navigation of a program between webpages. Web crawling allows a program to gather related data from multiple web pages and websites.

Consider `books.toscrape.com`, a site to practice web scraping that mimics a bookstore. The page `http://books.toscrape.com/catalogue/category/books/mystery_3/index.html` lists mystery books with overall ratings and review. More mystery books can be accessed by clicking on the `next` link. The following example demonstrates how to navigate between webpages to collect all of the mystery book titles.

<sup>2</sup>Though `requests` is not part of the standard library, it is recognized as a standard tool in the data science community. See <http://docs.python-requests.org/>.

```

def scrape_books(start_page = "index.html"):
    """ Crawl through http://books.toscrape.com and extract mystery titles"""

    # Initialize variables, including a regex for finding the 'next' link.
    base_url="http://books.toscrape.com/catalogue/category/books/mystery_3/"
    titles = []
    page = base_url + start_page          # Complete page URL.
    next_page_finder = re.compile(r"next") # We need this button.

    current = None

    for _ in range(2):
        while current == None: # Try downloading until it works.
            # Download the page source and PAUSE before continuing.
            page_source = requests.get(page).text
            time.sleep(1) # PAUSE before continuing.
            soup = BeautifulSoup(page_source, "html.parser")
            current = soup.find_all(class_="product_pod")

            # Navigate to the correct tag and extract title
            for book in current:
                titles.append(book.h3.a["title"])

            # Find the URL for the page with the next data.
            if "page-2" not in page:
                new_page = soup.find(string=next_page_finder).parent["href"]
                page = base_url + new_page # New complete page URL.
                current = None
    return titles

```

In this example, the `for` loop cycles through the pages of books, and the `while` loop ensures that each website page loads properly: if the downloaded `page_source` doesn't have a tag whose class is "product\_prod", the request is sent again. After recording all of the titles, the function locates the link to the next page. This link is stored in the HTML as a relative website path (page-2.html); the complete URL to the next day's page is the concatenation of the base URL `http://books.toscrape.com/catalogue/category/books/mystery_3/` with this relative link.

**Problem 1.** Modify `scrape_books()` so that it gathers the price for each fiction book and returns the mean price, in  $\mathcal{L}$ , of a fiction book.

An alternative approach that is often useful is to first identify the links to relevant pages, then scrape each of these page in succession. For example, the Federal Reserve releases quarterly data on large banks in the United States at `http://www.federalreserve.gov/releases/lbr`. The following function extracts the four measurements of total consolidated assets for JPMorgan Chase during 2004.

```
def bank_data():
    """Crawl through the Federal Reserve site and extract bank data."""
    # Compile regular expressions for finding certain tags.
    link_finder = re.compile(r"2004$")
    chase_bank_finder = re.compile(r"^JPMORGAN CHASE BK")

    # Get the base page and find the URLs to all other relevant pages.
    base_url="https://www.federalreserve.gov/releases/lbr/"
    base_page_source = requests.get(base_url).text
    base_soup = BeautifulSoup(base_page_source, "html.parser")
    link_tags = base_soup.find_all(name='a', href=True, string=link_finder)
    pages = [base_url + tag.attrs["href"] for tag in link_tags]

    # Crawl through the individual pages and record the data.
    chase_assets = []
    for page in pages:
        time.sleep(1)                # PAUSE, then request the page.
        soup = BeautifulSoup(requests.get(page).text, "html.parser")

        # Find the tag corresponding to Chase Banks's consolidated assets.
        temp_tag = soup.find(name="td", string=chase_bank_finder)
        for _ in range(10):
            temp_tag = temp_tag.next_sibling
        # Extract the data, removing commas.
        chase_assets.append(int(temp_tag.string.replace(',', '')))

    return chase_assets
```

**Problem 2.** Modify `bank_data()` so that it extracts the total consolidated assets (“Consol Assets”) for JPMorgan Chase, Bank of America, and Wells Fargo recorded each December from 2004 to the present. In a single figure, plot each bank’s assets against time. Be careful to keep the data sorted by date.

**Problem 3.** The Basketball Reference website at <https://www.basketball-reference.com> contains data on NBA athletes, including which player led different categories for each season. For the past ten seasons, identify which player had the most season points and find how many points they scored during that season. Return a list of triples consisting of the season, the player, and the points scored, (“season year”, “player name”, points scored).

## Asynchronously Loaded Content and User Interaction

Web crawling with the methods presented in the previous section fails under a few circumstances. First, many webpages use *JavaScript*, the standard client-side scripting language for the web, to load portions of their content *asynchronously*. This means that at least some of the content isn't initially accessible through the page's source code. Second, some pages require user interaction, such as clicking buttons which aren't links (`<a>` tags which contain a URL that can be loaded) or entering text into form fields (like search bars).

The *Selenium* framework provides a solution to both of these problems. Originally developed for writing unit tests for web applications, Selenium allows a program to open a web browser and interact with it in the same way that a human user would, including clicking and typing. It also has BeautifulSoup-esque tools for searching the HTML source of the current page.

### NOTE

Selenium requires an executable *driver* file for each kind of browser. The following examples use Google Chrome, but Selenium supports Firefox, Internet Explorer, Safari, Opera, and PhantomJS (a special browser without a user interface). See <https://seleniumhq.github.io/selenium/docs/api/py> or <http://selenium-python.readthedocs.io/installation.html> for installation instructions and driver download instructions.

To use Selenium, start up a browser using one of the drivers in `selenium.webdriver`. The browser has a `get()` method for going to different web pages, a `page_source` attribute containing the HTML source of the current page, and a `close()` method to exit the browser.

```
>>> from selenium import webdriver

# Start up a browser and go to example.com.
>>> browser = webdriver.Chrome()
>>> browser.get("https://www.example.com")

# Feed the HTML source code for the page into BeautifulSoup for processing.
>>> soup = BeautifulSoup(browser.page_source, "html.parser")
>>> print(soup.prettify())
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>
      Example Domain
    </title>
    <meta charset="utf-8"/>
    <meta content="text/html; charset=utf-8" http-equiv="Content-type"/>
  # ...

>>> browser.close()                                # Close the browser.
```

Selenium can deliver the HTML page source to BeautifulSoup, but it also has its own tools for finding tags in the HTML.

Method	Returns
<code>find_element_by_tag_name()</code>	The first tag with the given name
<code>find_element_by_name()</code>	The first tag with the specified <code>name</code> attribute
<code>find_element_by_class_name()</code>	The first tag with the given <code>class</code> attribute
<code>find_element_by_id()</code>	The first tag with the given <code>id</code> attribute
<code>find_element_by_link_text()</code>	The first tag with a matching <code>href</code> attribute
<code>find_element_by_partial_link_text()</code>	The first tag with a partially matching <code>href</code> attribute

Table 6.1: Methods of the `selenium.webdriver.Chrome` class.

Each of the `find_element_by_*`() methods returns a single object representing a *web element* (of type `selenium.webdriver.remote.webelement.WebElement`), much like a BeautifulSoup tag (of type `bs4.element.Tag`). If no such element can be found, a Selenium `NoSuchElementException` is raised. Each webdriver also has several `find_elements_by_*`() methods (elements, plural) that return a list of all matching elements, or an empty list if there are no matches.

Web element objects have methods that allow the program to interact with them: `click()` sends a click, `send_keys()` enters in text, and `clear()` deletes existing text. This functionality makes it possible for Selenium to interact with a website in the same way that a human would. For example, the following code opens up <https://www.google.com>, types “Python Selenium Docs” into the search bar, and hits enter.

```
>>> from selenium.webdriver.common.keys import Keys
>>> from selenium.common.exceptions import NoSuchElementException

>>> browser = webdriver.Chrome()
>>> try:
...     browser.get("https://www.google.com")
...     try:
...         # Get the search bar, type in some text, and press Enter.
...         search_bar = browser.find_element_by_name('q')
...         search_bar.clear()                # Clear any pre-set text.
...         search_bar.send_keys("Python Selenium Docs")
...         search_bar.send_keys(Keys.RETURN) # Press Enter.
...     except NoSuchElementException:
...         print("Could not find the search bar!")
...         raise
... finally:
...     browser.close()
... 
```

**Problem 4.** The arXiv (pronounced “archive”) is an online repository of scientific publications, hosted by Cornell University. Write a function that accepts a string to serve as a search query. Use Selenium to enter the query into the search bar of <https://arxiv.org> and press Enter. The resulting page has up to 50 links to the PDFs of technical papers that match the query. Gather these URLs, then continue to the next page (if there are more results) and continue gathering links until obtaining at most 150 URLs. Return the list of URLs.

#### NOTE

Using Selenium to access a page's source code is typically much safer, though slower, than using `requests.get()`, since Selenium waits for each web page to load before proceeding. For instance, the arXiv is a somewhat defensive about scrapers (<https://arxiv.org/help/robots>), but Selenium makes it possible to gather info from the website without offending the administrators.

**Problem 5.** *Project Euler* (<https://projecteuler.net>) is a collection of mathematical computing problems. Each problem is listed with an ID, a description/title, and the number of users that have solved the problem.

Using Selenium, BeautifulSoup, or both, for each of the (at least) 600 problems in the archive at <https://projecteuler.net/archives>, record the problem ID and the number of people who have solved it. Return a list of IDs, sorted from largest to smallest by the number of people who have solved them. That is, the first entry in the list should be the ID of the **most solved** problem, and the last entry in the list should be the ID of the **least solved** problem. (Hint: start by identifying the URLs to each archive page.)





# 7

## Pandas 1: Introduction

**Lab Objective:** *Though NumPy and SciPy are powerful tools for numerical computing, they lack some of the high-level functionality necessary for many data science applications. Python's pandas library, built on NumPy, is designed specifically for data management and analysis. In this lab, we introduce pandas data structures, syntax, and explore its capabilities for quickly analyzing and presenting data.*

Pandas are data structures used to manipulate data. They are built off of `ndarrays` and thereby carry many functionalities of arrays. A 1-d panda structure is called a **Series** and a 2-d panda structure is called a **DataFrame**. In the following section, we will read in data into a pandas **DataFrame** and manipulate the data using pandas commands.

### Reading Data

#### DataFrames

A **DataFrame** is a two-dimensional generalization of a NumPy array. While a NumPy array has rows and columns labeled by integers, dataframes can be labeled by any datatype. The row labels are collectively called the *index*, and the column labels are collectively called the *columns*. Unlike NumPy arrays, these labels are not necessarily integers, but can be set as any datatype. An individual column in a **DataFrame** object is one **Series**, or one-dimensional generalization of a NumPy array.

There are many ways to initialize a **DataFrame**. In the following code, we build a **DataFrame** out of a dictionary of **Series**.

```
>>> x = pd.Series(np.random.randn(4), ['a', 'b', 'c', 'd'])
>>> y = pd.Series(np.random.randn(5), ['a', 'b', 'd', 'e', 'f'])
>>> d = {"series 1": x, "series 2": y}
>>> df1 = pd.DataFrame(d)
>>> df1
```

	series 1	series 2
a	-0.365542	1.227960
b	0.080133	0.683523
c	0.737970	NaN
d	0.097878	-1.102835
e	NaN	1.345004

```
f      NaN    0.217523
```

The index of this `DataFrame` is the union of the index of `Series x` and that of `Series y`. The columns are given by the keys of the dictionary `d`. Since `x` doesn't have a label `e`, the value in row `e`, column 1 is `NaN`. This same reasoning explains the other missing values as well. We can recover the `Series x`, by using the function `dropna()` on the first column of the `DataFrame`:

```
>>> df1["series1"].dropna()
a    -0.365542
b     0.080133
c     0.737970
d     0.097878
Name: series 1, dtype: float64
```

### ACHTUNG!

A pandas `DataFrame` cannot be sliced in exactly the same way as a NumPy array. Notice how we just used `df1["series 1"]` to access a *column* of the the `DataFrame` `df1`. We will discuss this in more detail later on.

We can also initialize a `DataFrame` using a NumPy array, creating custom row and column labels.

```
>>> data = np.random.random((3, 4))
>>> pd.DataFrame(data, index=['A', 'B', 'C'], columns=np.arange(1, 5))
```

	1	2	3	4
A	0.065646	0.968593	0.593394	0.750110
B	0.803829	0.662237	0.200592	0.137713
C	0.288801	0.956662	0.817915	0.951016

3 rows      4 columns

If we don't specify the index or columns, the default is `np.arange(n)`, where `n` is either the number of rows or columns.

## Data I/O

The pandas library has functions that make importing and exporting data simple. The functions allow for a variety of file formats to be imported and exported, including CSV, Excel, HDF5, SQL, JSON, HTML, and pickle files.

Method	Description
<code>to_csv()</code>	Write the index and entries to a CSV file
<code>to_json()</code>	Convert the object to a JSON string
<code>to_pickle()</code>	Serialize the object and store it in an external file
<code>to_sql()</code>	Write the object data to an open SQL database

Table 7.1: Methods for exporting data in a pandas `Series` or `DataFrame`.

The CSV (comma separated values) format is a simple way of storing tabular data in plain text. Because CSV files are one of the most popular file formats for exchanging data, we will explore the `read_csv()` function in more detail. To learn to read other types of file formats, see the online pandas documentation. To read a CSV data file into a **DataFrame**, call the `read_csv()` function with the path to the CSV file, along with the appropriate keyword arguments. Below we list some of the most important keyword arguments:

- **delimiter:** The character that separates data fields. It is often a comma or a whitespace character.
- **header:** The row number (starting at 0) in the CSV file that contains the column names.
- **index\_col:** The column in the CSV file that is the index for the **DataFrame**.
- **skiprows:** If an integer  $n$ , skip the first  $n$  rows of the file, and then start reading in the data. If a list of integers, skip the specified rows.
- **names:** If the CSV file does not contain the column names, or you wish to use other column names, specify them in a list.

**Problem 1.** Read in the file `crime_data.csv` as a pandas object. The file contains data on types of crimes in the U.S. from 1960 to 2016. Set the index as the column 'Year'. Skip any lines that do not contain data.

## Data Manipulation

### Basic Data Access

While subsets of data can be accessed in **DataFrames** using slicing, similar to a NumPy array, subsets can also be selected using the `iloc` and `loc` indexers. Accessing **Series** and **DataFrame** objects using these indexing operations is more efficient than slicing, because the bracket indexing has to check many cases before it can determine how to slice the data structure. Using `loc/iloc` explicitly, bypasses the extra checks. The `loc` index selects rows and columns based on their labels, while the `iloc` method selects them based on their integer position.

=====

## Reading Data

### DataFrames

Pandas are data structures used to manipulate data. They are built off of **ndarrays** and thereby carry many functionalities of arrays. A 1-d panda structure is called a **Series** and a 2-d panda structure is called a **DataFrame**. A **DataFrame** is a two-dimensional generalization of a NumPy array. While a NumPy array has rows and columns labeled by integers, dataframes can be labeled by any datatype. The row labels are collectively called the *index*, and the column labels are collectively called the *columns*. An individual column in a **DataFrame** object is one **Series**, or one-dimensional generalization of a NumPy array. There are many ways to initialize a **DataFrame**. In the following code, we build a **DataFrame** out of a dictionary of **Series**.

```
>>> x = pd.Series(np.random.randn(4), ['a', 'b', 'c', 'd'])
>>> y = pd.Series(np.random.randn(5), ['a', 'b', 'd', 'e', 'f'])
>>> d = {"series 1": x, "series 2": y}
>>> df1 = pd.DataFrame(d)
>>> df1
```

	series 1	series 2
a	-0.365542	1.227960
b	0.080133	0.683523
c	0.737970	NaN
d	0.097878	-1.102835
e	NaN	1.345004
f	NaN	0.217523

The index of this `DataFrame` is the union of the index of `Series x` and that of `Series y`. The columns are given by the keys of the dictionary `d`. Since `x` doesn't have a label `e`, the value in row `e`, column 1 is `NaN`. This same reasoning explains the other missing values as well. We can recover the `Series x`, by using the function `dropna()` on the first column of the `DataFrame`:

```
>>> df1["series1"].dropna()
a    -0.365542
b     0.080133
c     0.737970
d     0.097878
Name: series 1, dtype: float64
```

### ACHTUNG!

A pandas `DataFrame` cannot be sliced in exactly the same way as a NumPy array. Notice how we just used `df1["series 1"]` to access a *column* of the the `DataFrame` `df1`. We will discuss this in more detail later on.

We can also initialize a `DataFrame` using a NumPy array, creating custom row and column labels.

```
>>> data = np.random.random((3, 4))
>>> pd.DataFrame(data, index=['A', 'B', 'C'], columns=np.arange(1, 5))
```

	1	2	3	4
A	0.065646	0.968593	0.593394	0.750110
B	0.803829	0.662237	0.200592	0.137713
C	0.288801	0.956662	0.817915	0.951016

3 rows      4 columns

If we don't specify the index or columns, the default is `np.arange(n)`, where `n` is either the number of rows or columns.

## Data I/O

The pandas library has functions that make importing and exporting data simple. The functions allow for a variety of file formats to be imported and exported, including CSV, Excel, HDF5, SQL, JSON, HTML, and pickle files.

Method	Description
<code>to_csv()</code>	Write the index and entries to a CSV file
<code>to_json()</code>	Convert the object to a JSON string
<code>to_pickle()</code>	Serialize the object and store it in an external file
<code>to_sql()</code>	Write the object data to an open SQL database

Table 7.2: Methods for exporting data in a pandas **Series** or **DataFrame**.

The CSV (comma separated values) format is a simple way of storing tabular data in plain text. Because CSV files are one of the most popular file formats for exchanging data, we will explore the `read_csv()` function in more detail. To learn to read other types of file formats, see the online pandas documentation. To read a CSV data file into a **DataFrame**, call the `read_csv()` function with the path to the CSV file, along with the appropriate keyword arguments. Below we list some of the most important keyword arguments:

- **delimiter:** The character that separates data fields. It is often a comma or a whitespace character.
- **header:** The row number (starting at 0) in the CSV file that contains the column names.
- **index\_col:** The column in the CSV file that is the index for the **DataFrame**.
- **skiprows:** If an integer  $n$ , skip the first  $n$  rows of the file, and then start reading in the data. If a list of integers, skip the specified rows.
- **names:** If the CSV file does not contain the column names, or you wish to use other column names, specify them in a list.

**Problem 2.** Read in the file `crime_data.csv` as a pandas object. The file contains data on types of crimes in the U.S. from 1960 to 2016. Set the index as the column 'Year'. Skip any lines that do not contain data.

## Data Manipulation

### Basic Data Access

While subsets of data can be accessed in **DataFrames** using slicing, similar to a NumPy array, subsets can also be selected using the `iloc` and `loc` indexers. Accessing **Series** and **DataFrame** objects using these indexing operations is more efficient than slicing because the bracket indexing has to check many cases before it can determine how to slice the data structure. Using `loc/iloc` explicitly, bypasses the extra checks. The `loc` index selects rows and columns based on their labels, while the `iloc` method selects them based on their integer position.

```
>>> df = pd.DataFrame(np.random.randn(4,2), index=['a', 'b', 'c', 'd'], columns←
    =['I', 'II'])
>>> df
```

	I	II
a	-0.154878	-1.097156
b	-0.948226	0.585780
c	0.433197	-0.493048
d	-0.168612	0.999194

```
[4 rows x 2 columns]

>>> # select rows a and c, column II
>>> df.loc[['a','c'], 'II']
```

	II
a	-1.097156
c	-0.493048

```
Name: II, dtype: float64

>>> # select last two rows, first column
>>> df.iloc[-2:, 0]
```

	I
c	0.433197
d	-0.168612

```
Name: I, dtype: float64
```

A column of a `DataFrame` may also be accessed using simple square brackets and the name of the column, or alternatively by treating the label as an object:

```
>>> # get second column of df
>>> df['II']          # or, equivalently, df.II
```

	II
a	-1.097156
b	0.585780
c	-0.493048
d	0.999194

```
Name: II, dtype: float64
```

Both of these techniques for getting subsets of the data may also be used to set subsets of the data:

```
>>> # set second columns to zeros
>>> df['II'] = 0
>>> df['II']
```

	II
a	0
b	0
c	0
d	0

```
Name: II, dtype: int64
```

```
>>> # add additional column of ones
>>> df['III'] = 1
>>> df
```

	I	II	III
a	-0.154878	0	1
b	-0.948226	0	1
c	0.433197	0	1
d	-0.168612	0	1

Additionally, pandas have functions such as `describe()`, `head()`, and `tail()` which can help to view and analyze data.

```
# Use head to see the first n rows
>>> df.head(n=2)
```

	I	II	III
a	-0.154878	0	1
b	-0.948226	0	1

```
# Use describe to find stats about the dataframe
>>> df.describe()
```

	I	II
count	4.000000	4.000000
mean	-0.209630	-0.001307
std	0.566695	0.964083
min	-0.948226	-1.097156
25%	-0.363515	-0.644075
50%	-0.161745	0.046366
75%	-0.007895	0.689133
max	0.433197	0.999194

===== Other methods used for manipulating **DataFrame** and **Series** panda structures can be found in Table 7.5.

Method	Description
<code>append()</code>	Concatenate two or more <b>Series</b> .
<code>drop()</code>	Remove the entries with the specified label or labels
<code>drop_duplicates()</code>	Remove duplicate values
<code>dropna()</code>	Drop null entries
<code>fillna()</code>	Replace null entries with a specified value or strategy
<code>reindex()</code>	Replace the index
<code>sample()</code>	Draw a random entry
<code>shift()</code>	Shift the index
<code>unique()</code>	Return unique values

Table 7.3: Methods for managing or modifying data in a pandas **Series** or **DataFrame**.

## Basic Data Manipulation

Because the primary pandas data structures are subclasses of the `ndarray`, most NumPy functions work with pandas structure. For example, basic vector operations work just fine:

```
>>> x = pd.Series(np.random.randn(4), index=['a', 'b', 'c', 'd'])
>>> y = pd.Series(np.random.randn(5), index=['a', 'b', 'd', 'e', 'f'])
>>> x**2
a    1.710289
b    0.157482
c    0.540136
d    0.202580
dtype: float64
>>> z = x + y
>>> z
a    0.123877
b    0.278435
c         NaN
d   -1.318713
e         NaN
f         NaN
dtype: float64
```

It may also be useful to transpose `DataFrames`, re-order the columns or rows, or sort according to a given column.

```
# re-order columns
>>> df.reindex(columns = ['II', 'I'])
      II      I
a -1.097156 -0.154878
b  0.585780 -0.948226
c -0.493048  0.433197
d  0.999194 -0.168612

# sort descending according to column 'II'
>>> df.sort_values('II', ascending=False)
      I      II
d -0.168612  0.999194
b -0.948226  0.585780
c  0.433197 -0.493048
a -0.154878 -1.097156

[4 rows x 2 columns]
```

**Problem 3.** Create a new column `Rate` which contains the crime rate for each year. Change each column referring to a specific crime to a percentage in relation to the total amount of crime that year. Use this updated table to answer the following questions:



1. What is the average crime rate? Is the crime rate generally above or below the median?
2. On average, what is the least common crime?
3. Identify two crimes which have close percentages. Do these crimes have any kind of correlation from year to year? If so, what is the correlation?

## Dealing with Missing Data

Missing data is a ubiquitous problem in data science. Fortunately, pandas is particularly well-suited to handling missing and anomalous data. As we have already seen, the pandas default for a missing value is `NaN`. In basic arithmetic operations, if one of the operands is `NaN`, then the output is also `NaN`. If we are not interested in the missing values, we can simply drop them from the data altogether:

```
>>> x = pd.Series(np.arange(5))
>>> y = pd.Series(np.random.randn(5))
>>> x.iloc[3] = np.nan
>>> x + y
0    0.731521
1    0.623651
2    2.396344
3         NaN
4    3.351182
dtype: float64

=====

>>> (x + y).dropna()
0    0.731521
1    0.623651
2    2.396344
4    3.351182
dtype: float64
```

This is not always the desired behavior, however. Missing data could actually correspond to some default value, such as zero. In this case, we can replace all instances of `NaN` with a specified value:

```
>>> # fill missing data with 0
>>> (x + y).fillna(0)
0    0.731521
1    0.623651
2    2.396344
3    0.000000
4    3.351182
dtype: float64
```

When dealing with missing data, make sure you are aware of the behavior of the pandas functions you are using. For example, `sum()` and `mean()` ignore NaN values in the computation.

## Data Manipulation with Series

A **Series** object, or column of a **DataFrame**, has all of the advantages of a NumPy array, including entry-wise arithmetic, plus a few additional features (see Table 7.4).

Method	Returns
<code>abs()</code>	Object with absolute values taken (of numerical data)
<code>argmax()</code>	The index label of the maximum value
<code>argmin()</code>	The index label of the minimum value
<code>count()</code>	The number of non-null entries
<code>cumprod()</code>	The cumulative product over an axis
<code>cumsum()</code>	The cumulative sum over an axis
<code>max()</code>	The maximum of the entries
<code>mean()</code>	The average of the entries
<code>median()</code>	The median of the entries
<code>min()</code>	The minimum of the entries
<code>mode()</code>	The most common element(s)
<code>prod()</code>	The product of the elements
<code>sum()</code>	The sum of the elements
<code>var()</code>	The variance of the elements

Table 7.4: Numerical methods of the **Series** and **DataFrame** pandas classes.

**Series** are more useful than NumPy arrays when dealing with data primarily because of their index. For example, a **Series** can be indexed by time with a pandas **DatetimeIndex**, an index with date and/or time values. The usual way to create this kind of index is with `pd.date_range()`.

```
# Make an index of the first three days in July 2000.
>>> pd.date_range("7/1/2000", "7/3/2000", freq='D')
DatetimeIndex(['2000-07-01', '2000-07-02', '2000-07-03'],
              dtype='datetime64[ns]', freq='D')
```

**Problem 4.** Answer the following questions using the file `crime_data.csv` and the functions described above. Support your answers with data from the file.

1. In what year did the majority of crimes hit an all-time high?
2. Which decade had the greatest amount of total crime?
3. Which crime was committed most during mentioned decade?

=====

Method	Description
<code>append()</code>	Concatenate two or more <b>Series</b> .
<code>drop()</code>	Remove the entries with the specified label or labels
<code>drop_duplicates()</code>	Remove duplicate values
<code>dropna()</code>	Drop null entries
<code>fillna()</code>	Replace null entries with a specified value or strategy
<code>reindex()</code>	Replace the index
<code>sample()</code>	Draw a random entry
<code>shift()</code>	Shift the index
<code>unique()</code>	Return unique values

Table 7.5: Methods for managing or modifying data in a pandas **Series** or **DataFrame**.

## SQL Operations in pandas

The **DataFrame**, being a tabular data structure, bears an obvious resemblance to a typical relational database table. SQL is the standard for working with relational databases, and in this section we will explore how pandas accomplishes some of the same tasks as SQL. The SQL-like functionality of pandas is one of its biggest advantages, since it can eliminate the need to switch between programming languages for different tasks. Within pandas, we can handle both the querying *and* data analysis.

For the examples below, we will use the following data:

```
>>> name = ['Mylan', 'Regan', 'Justin', 'Jess', 'Jason', 'Remi', 'Matt', '↵
Alexander', 'JeanMarie']
>>> sex = ['M', 'F', 'M', 'F', 'M', 'F', 'M', 'M', 'F']
>>> age = [20, 21, 18, 22, 19, 20, 20, 19, 20]
>>> rank = ['Sp', 'Se', 'Fr', 'Se', 'Sp', 'J', 'J', 'J', 'Se']
>>> ID = range(9)
>>> aid = ['y', 'n', 'n', 'y', 'n', 'n', 'n', 'y', 'n']
>>> GPA = [3.8, 3.5, 3.0, 3.9, 2.8, 2.9, 3.8, 3.4, 3.7]
>>> mathID = [0, 1, 5, 6, 3]
>>> mathGd = [4.0, 3.0, 3.5, 3.0, 4.0]
>>> major = ['y', 'n', 'y', 'n', 'n']
>>> studentInfo = pd.DataFrame({'ID': ID, 'Name': name, 'Sex': sex, 'Age': age, ↵
'Class': rank})
>>> otherInfo = pd.DataFrame({'ID': ID, 'GPA': GPA, 'Financial_Aid': aid})
>>> mathInfo = pd.DataFrame({'ID': mathID, 'Grade': mathGd, 'Math_Major': major ↵
})
```

Before querying our data, it is important to know some of its basic properties, such as number of columns, number of rows, and the datatypes of the columns. This can be done by simply calling the `info()` method on the desired **DataFrame**:

```
>>> mathInfo.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 5 entries, 0 to 4
Data columns (total 3 columns):
Grade      5 non-null float64
ID         5 non-null int64
```

```
Math_Major      5 non-null object
dtypes: float64(1), int64(1), object(1)
```

The method `isin()` is a useful way to find certain values in a `DataFrame`. It compares the input (a list, dictionary, or `Series`) to the `DataFrame` and returns a boolean `Series` showing whether or not the values match. One can then use this boolean array to select appropriate locations. Now let's look at the pandas equivalent of some SQL SELECT statements.

```
>>> # SELECT ID, Age FROM studentInfo
>>> studentInfo[['ID', 'Age']]
   ID  Age
0    0   20
1    1   21
2    2   18
3    3   22
4    4   19
5    5   20
6    6   20
7    7   19
8    8   29

>>> # SELECT ID, GPA FROM otherInfo WHERE Financial_Aid = 'y'
>>> otherInfo[otherInfo['Financial_Aid']=='y']['ID', 'GPA']
   ID  GPA
0    0  3.8
3    3  3.9
7    7  3.4

>>> # SELECT Name FROM studentInfo WHERE Class = 'J' OR Class = 'Sp'
>>> studentInfo[studentInfo['Class'].isin(['J', 'Sp'])['Name']]
0      Mylan
4      Jason
5       Remi
6       Matt
7  Alexander
Name: Name, dtype: object
```

**Problem 5.** Using panda commands, create a table of the crime rates in years where Property crimes were less than 90% of the total crime and burglary crimes were more than 20%, or property crimes were less than 90% while violent crimes were more than 20%.

Next, let's look at JOIN statements. In pandas, this is done with the `merge` function, which takes as arguments the two `DataFrame` objects to join, as well as keyword arguments specifying the column on which to join, along with the type (left, right, inner, outer).

```
>>> # SELECT * FROM studentInfo INNER JOIN mathInfo ON studentInfo.ID = ←
      mathInfo.ID
```

```
>>> pd.merge(studentInfo, mathInfo, on='ID') # INNER JOIN is the default
  Age Class  ID  Name Sex  Grade Math_Major
0   20    Sp   0  Mylan  M    4.0          y
1   21    Se   1  Regan  F    3.0          n
2   22    Se   3   Jess  F    4.0          n
3   20     J   5   Remi  F    3.5          y
4   20     J   6   Matt  M    3.0          n
[5 rows x 7 columns]
```

```
>>> # SELECT GPA, Grade FROM otherInfo FULL OUTER JOIN mathInfo ON otherInfo.ID↵
    = mathInfo.ID
>>> pd.merge(otherInfo, mathInfo, on='ID', how='outer')[['GPA', 'Grade']]
   GPA  Grade
0  3.8    4.0
1  3.5    3.0
2  3.0   NaN
3  3.9    4.0
4  2.8   NaN
5  2.9    3.5
6  3.8    3.0
7  3.4   NaN
8  3.7   NaN
[9 rows x 2 columns]
```

**Problem 6.** Read in the files `final_accidents2.pickle` and `final_drivers.pickle` using the function `read_pickle()`. The accidents file contains data on various accidents that occurred in the U.S. and their causes. The drivers data has data on various tickets given to drivers, how many tickets given, and what the ticket is for. The following table shows the meanings of the codes in the data.

Code	Description
DRUNK_DR	0-99 - Number of drunk drivers
WEATHER	0 - Unknown 1 - Clear 2 - Rain 3 - Sleet, Hail 4 - Snow 5 - Fog, Smog, Smoke 6 - Severe Crosswinds 7 - Blowing Sand, Soil, and Dirt 8 - Other 10 - Cloudy 11 - Blowing Snow 12 - Freezing Rain or Drizzle 98 - Not Reported 99 - Unknown
SP	0 - Not Speeding 1 - Speeding 8 - Not Reported 9 - Unknown
FATALS	0-99 - Number of fatalities
VEH_NO	Assigned number of motor vehicle
DRINKING	0 - Alcohol Not Involved 1 - Alcohol Involved 8 - Not Reported 9 - Unknown
SPEEDREL	0 - No 1 - Yes 2 - Yes, Racing 3 - Yes, Exceeded Speed Limit 4 - Yes, Too Fast for Conditions 5 - Yes, Specifics Unknown 8 - No Driver Present/Unknown if Driver Present 9 - Unknown

Table 7.6: Codes for FARS data.

Create a `DataFrame` containing the Case Number, Age, whether the accident was caused by a drunk driver, whether the driver had DUIs, and if the accident was fatal, using an inner join operation. Create a second `DataFrame` containing Case Number, Age, and fatalities using an inner join operation. In this `DataFrame`, also include the speed column from each dataframe.

**Problem 7.** Use the four panda structures created in Problem 6 to answer the following questions. Make sure to support your claims with appropriate and presentable data.

1. What is the most common kind of weather involved with accidents?
2. How often did speeding get reported in accidents?
3. Of accidents where the type of speeding is known, what is the most common type of speeding in accidents?
4. On average, how many drunk drivers are involved in an accident?
5. What month has the most fatalities each year?







## Pandas 2: Plotting

**Lab Objective:** *Clear, insightful visualizations are a crucial part of data analysis. To facilitate quick data visualization, pandas includes several tools that wrap around matplotlib. These tools make it easy to compare different parts of a data set, explore the data as a whole, and spot patterns and correlations in the data.*

### Overview of Plotting Tools

The main tool for visualization in pandas is the `plot()` method of the `Series` and `DataFrame`. The method has a keyword argument `kind` that specifies the type of plot to draw. The valid options for `kind` are detailed below.

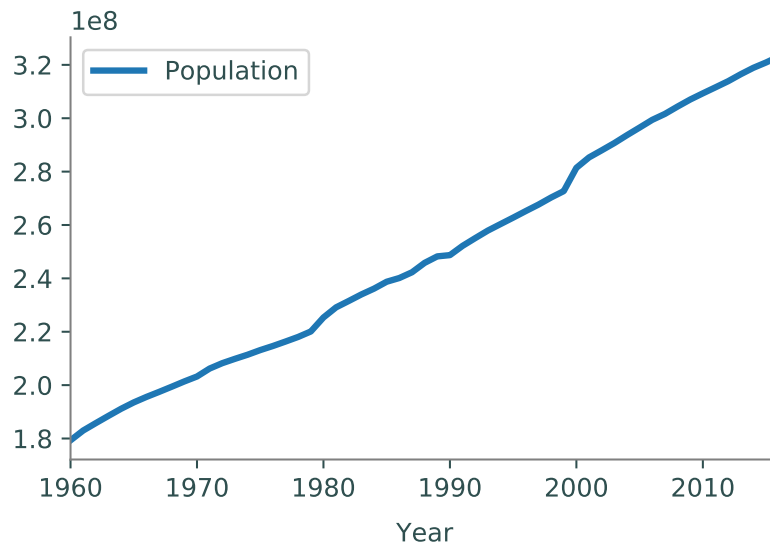
Plot Type	plot() ID	Uses and Advantages
Line plot	"line"	Show trends ordered in data; easy to compare multiple data sets
Scatter plot	"scatter"	Compare exactly two data sets, independent of ordering
Bar plot	"bar", "barh"	Compare categorical or sequential data
Histogram	"hist"	Show frequencies of one set of values, independent of ordering
Box plot	"box"	Display min, median, max, and quartiles; compare data distributions
Hexbin plot	"hexbin"	2D histogram; reveal density of cluttered scatter plots

Table 8.1: Uses for the `plot()` method of the pandas `Series` and `DataFrame`. The plot ID is the value of the keyword argument `kind`. That is, `df.plot(kind="scatter")` creates a scatter plot. The default `kind` is "line".

The `plot()` method calls `plt.plot()`, `plt.hist()`, `plt.scatter()`, and other matplotlib plotting function, but it also assigns axis labels, tick marks, legends, and a few other things based on the index and the data. Most calls to `plot()` specify the `kind` of plot and which `Series` (column) to use as the `x` and `y` axes. By default, the `index` of the `Series` or `DataFrame` is used for the `x` axis.

```
>>> import pandas as pd
>>> from matplotlib import pyplot as plt

>>> crime = pd.read_csv("crime_data.csv", index_col="Year")
>>> crime.plot(y="Population") # Plot population against the index (years).
```



In this case, the call to the `plot()` method is essentially equivalent to the following code.

```
>>> plt.plot(crime.index, crime["Population"], label="Population")
>>> plt.xlabel(crime.index.name)
>>> plt.xlim(min(crime.index), max(crime.index))
>>> plt.legend(loc="best")
```

The `plot()` method also takes in many keyword arguments for matplotlib plotting and annotation functions. For example, setting `legend=False` disables the legend, providing a value for `title` sets the figure title, `grid=True` turns a grid on, and so on. For more customizations, see <https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.plot.html>.

## Visualizing an Entire Data Set

A good way to start analyzing an unfamiliar data set is to visualize as much of the data as possible to determine which parts are most important or interesting. For example, since the columns in a `DataFrame` share the same index, the columns can all be graphed together using the index as the *x*-axis. The `plot()` method attempts by default to plot **every** Series in a `DataFrame`. This is especially useful with sequential data, like the crime data set.

```
# Plot all columns together against the index.
>>> crime.plot(linewidth=1)

# Plot all columns together except for 'Population' and 'Total'.
>>> crime.drop(["Population", "Total"], axis=1).plot(linewidth=1)
```

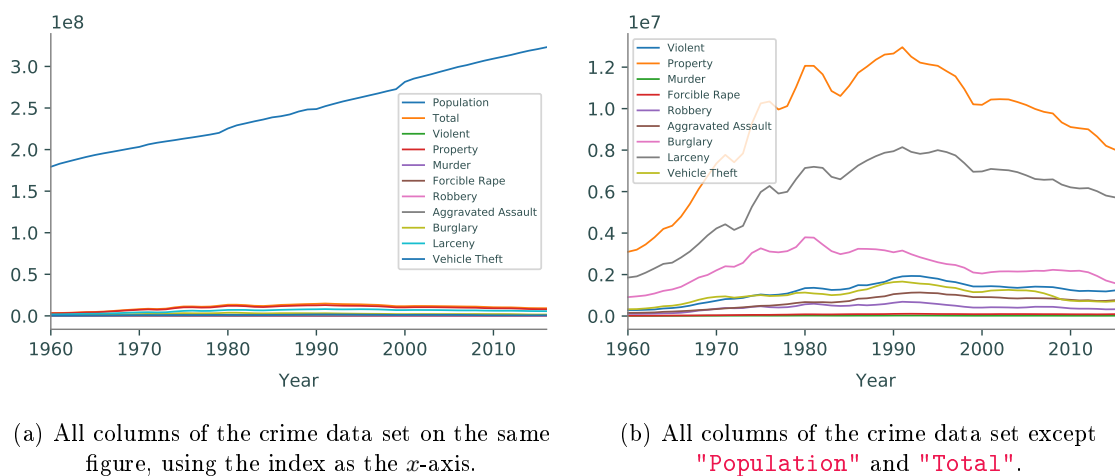


Figure 8.1

While plotting every **Series** at once can give an overview of all the data, the resulting plot is often difficult for the reader to understand. For example, the crime data set has 11 columns, so the resulting figure, Figure 8.1a, is fairly cluttered.

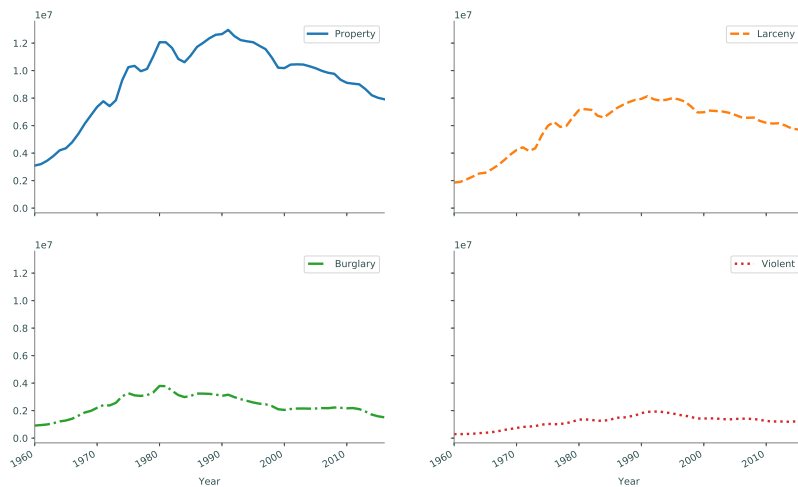
One way to declutter a visualization is to examine less data. Notice that the **Population** column has values much bigger than the other columns. This is because the "Population" column is on a completely different scale than the others. Dropping this column, as well as 'Total', gives a better overview of the data, shown in Figure 8.1b.

### ACHTUNG!

The "Population" column radically differs from the other columns because it has **different units of measure**: population is measured by “number of people,” but all other columns are measured in “number of crimes.” Be careful not to plot parts of a data set together if those parts don’t have the same units or are otherwise incomparable.

Another way to declutter a plot is to use subplots. To quickly plot several columns in separate subplots, use `subplots=True` and specify a shape tuple as the `layout` for the plots. Subplots automatically share the same  $x$ -axis. Set `sharey=True` to force them to share the same  $y$ -axis as well.

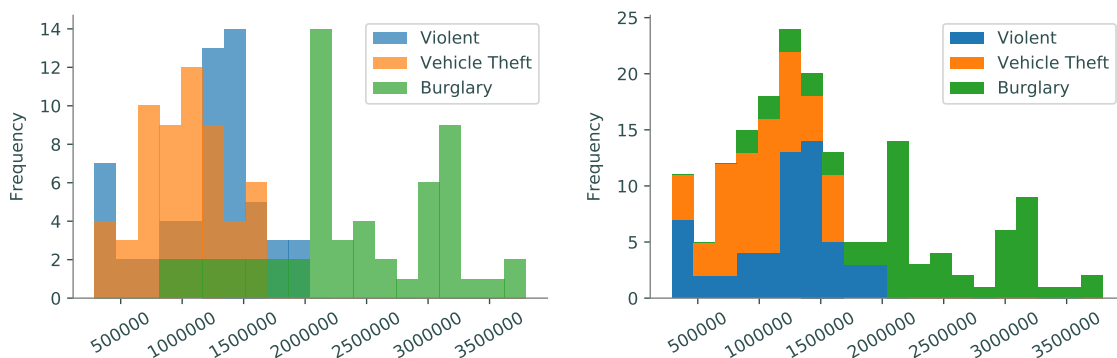
```
>>> crime.plot(y=["Property", "Larceny", "Burglary", "Violent"],
...             subplots=True, layout=(2,2), sharey=True,
...             style=['-', '--', '-.', ':'])
```



As mentioned previously, the `plot()` method can be used to plot different kinds of plots. One kind of plot that is possible with the `plot()` method is a histogram. However, plots made by the `plot()` method share an *x*-axis by default, so histograms turn out poorly whenever there are columns with very different data ranges or when more than one column is plotted at once. Thus, histograms are good for examining the distribution of a **single** column in a data set.

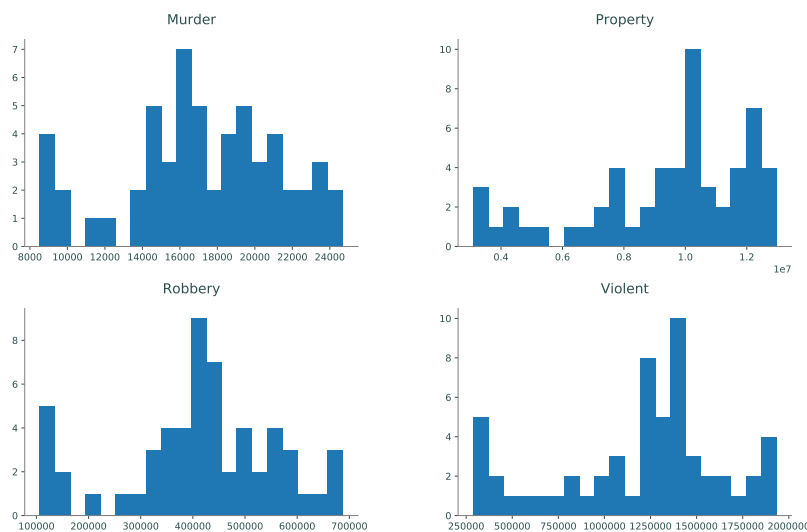
```
# Plot three histograms together.
>>> crime.plot(kind="hist", y=["Violent", "Vehicle Theft", "Burglary"],
...             bins=20, alpha=.7, rot=30)

# Plot three histograms, stacking one on top of the other.
>>> crime.plot(kind="hist", y=["Violent", "Vehicle Theft", "Burglary"],
...             bins=20, stacked=True, rot=30)
```



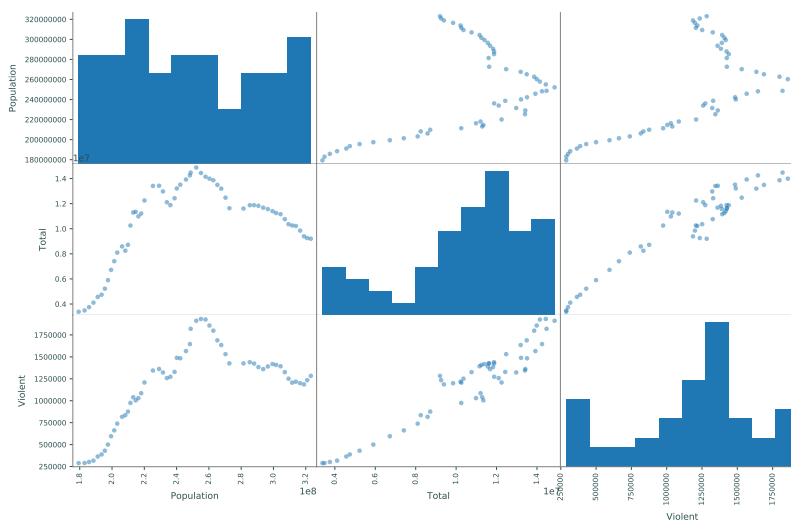
For histograms, use the `hist()` method of the `DataFrame` instead of the `plot()` method. Specify the number of bins with the `bins` parameter. Make sure to choose a number of bins that accurately represents the data; the wrong number of bins can create a misleading or uninformative visualization.

```
>>> crime[["Violent", "Murder", "Robbery", "Property"]].hist(grid=False, bins=20)
```



Another useful visualization when beginning to understand a data set is a scatter matrix. The function `pd.plotting.scatter_matrix()` produces a table of plots where each column is plotted against each other column in separate scatter plots. The plots on the diagonal, instead of plotting a column against itself, displays a histogram of that column. This provides a very quickly method to do an initial analysis of the correlation between different columns.

```
>>> pd.plotting.scatter_matrix(crime[["Population", "Total", "Violent"]])
```



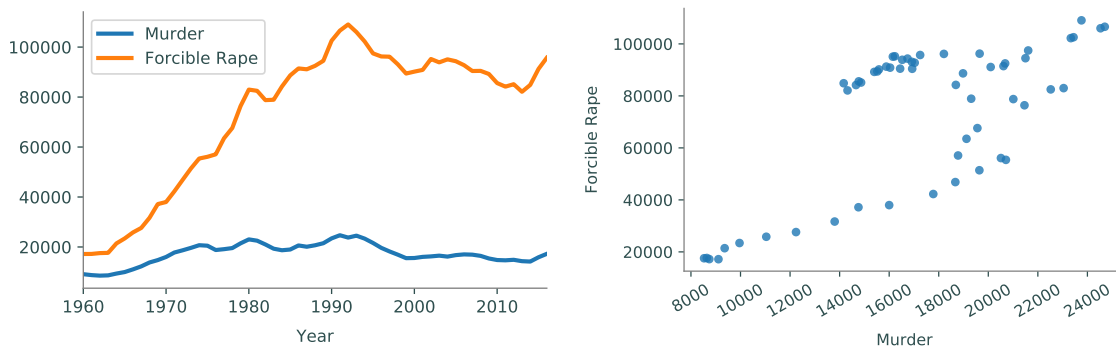
**Problem 1.** Create 3 visualizations for the data in `final_accidents2.pickle` and `final_drivers.pickle`. Make one of the visualizations a histogram and one visualization a scatter matrix. The visualizations should be well labeled and easy to understand.

## Patterns and Correlations

After visualizing the entire data set initially, a good next step is to closely compare related parts of the data. This can be done with different types of visualizations. For example, Figure 8.1b suggests that the "Murder" and "Forcible Rape" columns are roughly on the same scale. Since this data is sequential (indexed by time), start by plotting these two columns against the index. Next, create a scatter plot of one of the columns versus the other to investigate correlations that are independent of the index. Unlike other types of plots, using `kind="scatter"` requires both an `x` and a `y` column as arguments.

```
# Plot 'Murder' and 'Forcible Rape' as lines against the index.
>>> crime.plot(y=["Murder", "Forcible Rape"])

# Make a scatter plot of 'Murder' against 'Forcible Rape', ignoring the index.
>>> crime.plot(kind="scatter", x="Murder", y="Forcible Rape", alpha=.8, rot=30)
```



What do these graphs show about the data? First of all, rape is more common than murder. Second, rates of rape appear to be steadily increased from the mid 1960's to the mid 1990's before leveling out, while murder rates stay relatively constant. The disparity between rape and murder is confirmed in the scatter plot: the clump of data points at about 15,000 murders and 90,000 rapes shows that there have been many years where rape was relatively high while murder was somewhat low.

### ACHTUNG!

While analyzing data, especially while searching for patterns and correlations, **always** ask yourself if the data makes sense and is trustworthy. What lurking variables could have influenced the data measurements as they were being gathered?

The crime data set is somewhat suspect in this regard. The murder rate is likely accurate, since murder is conspicuous and highly reported, but what about the rape rate? Are the number of rapes increasing, or is the percentage of rapes being reported increasing? (It's probably both!) Be careful about drawing conclusions for sensitive or questionable data.

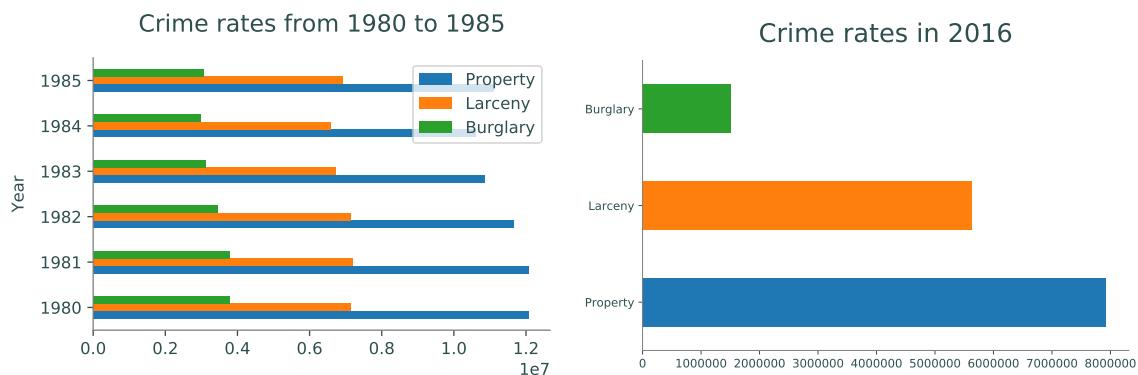
## Bar Graphs

Different types of graphs help to examine different kinds of patterns. Note that Figure 8.1b reveals some general patterns relative to time. For instance, there seems to be a small bump in each type of crime in the early 1980's. Slicing the entries from 1980 to 1985 provides a closer look. Since there are only a few entries, we can treat the data as categorical and make a bar chart.

As a general rule, horizontal bar charts (`kind="hbar"`) are better than the default vertical bar charts (`kind="bar"`) because most humans can detect horizontal differences more easily than vertical differences. If the labels are too long to fit on a normal figure, use `plt.tight_layout()` to adjust the plot boundaries to fit the labels in.

```
# Plot 'Property' and 'Larceny' rates from 1980 to 1985.
>>> crime.loc[1980:1985, ["Property", "Larceny", "Burglary"]].plot(kind="barh",
...                                                                    title="Crime rates from 1980 to 1985")

# Plot the most recent year's crime rates for comparison.
>>> crime.iloc[-1][["Property", "Larceny", "Burglary"]].plot(kind="barh",
...                                                            title="Crime rates in 2016", color=["C0", "C1", "C2"])
>>> plt.tight_layout()
```



**Problem 2.** Use visualizations to identify if trends exist between FATALS and the following variables:

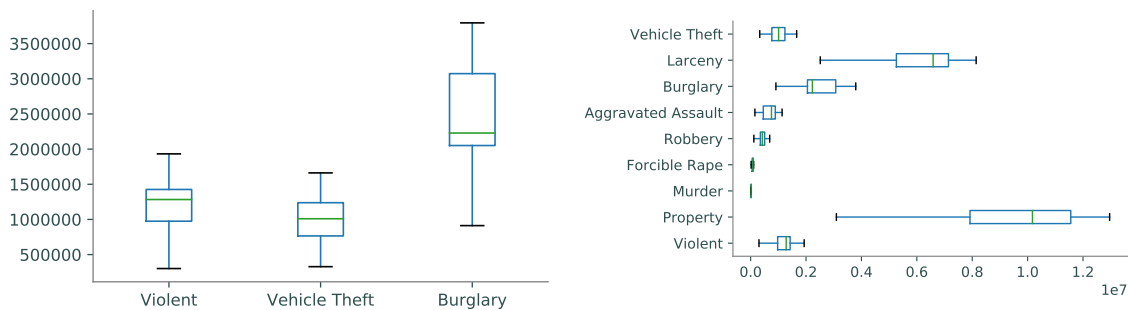
1. WEATHER
2. DRINKING
3. SP

## Distributional Visualizations

While histograms are good at displaying the distributions on one column, a different visualization is needed to show the distribution of an entire set. A *box plot* (sometimes called a “cat-and-whisker” plot) shows the five number summary: the minimum, first quartile, median, third quartile, and maximum of the data. Box plots are useful for comparing the distributions of relatable data. However, box plots are a basic summary, meaning that they are susceptible to miss important information such as how many points were in each distribution.

```
# Compare the distributions of three columns.
>>> crime.plot(kind="box", y=["Violent", "Vehicle Theft", "Burglary"])

# Compare the distributions of all columns but 'Population' and 'Total'.
>>> crime.drop(["Population", "Total"], axis=1).plot(kind="box", vert=False)
```



## Hexbin Plots

A scatter plot is essentially a plot of samples from the joint distribution of two columns. However, scatter plots can be uninformative for large data sets when the points in a scatter plot are closely clustered. *Hexbin plots* solve this problem by plotting point density in hexagonal bins—essentially creating a 2-dimensional histogram.

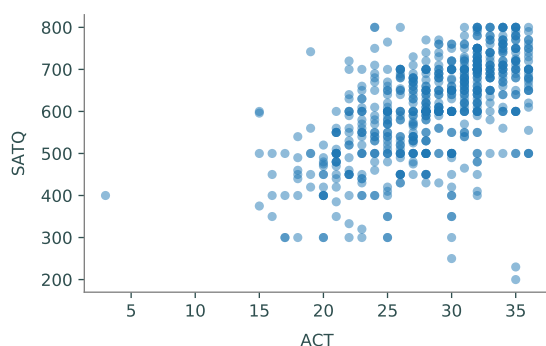
The file `sat_act.csv` contains 700 self reported scores on the SAT Verbal, SAT Quantitative and ACT, collected as part of the Synthetic Aperture Personality Assessment (SAPA) web based personality assessment project. The obvious question with this data set is “how correlated are ACT and SAT scores?” The scatter plot of ACT scores versus SAT Quantitative scores, Figure 8.6a, is highly cluttered, even though the points have some transparency. A hexbin plot of the same data, Figure 8.6b, reveals the **frequency** of points in binned regions.



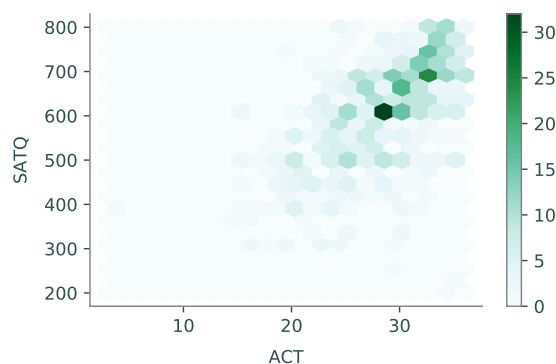
```
>>> satact = pd.read_csv("sat_act.csv", index_col="ID")
>>> list(satact.columns)
['gender', 'education', 'age', 'ACT', 'SATV', 'SATQ']

# Plot the ACT scores against the SAT Quant scores in a regular scatter plot.
>>> satact.plot(kind="scatter", x="ACT", y="SATQ", alpha=.8)

# Plot the densities of the ACT vs. SATQ scores with a hexbin plot.
>>> satact.plot(kind="Hexbin", x="ACT", y="SATQ", gridsize=20)
```



(a) ACT vs. SAT Quant scores.



(b) Frequency of ACT vs. SAT Quant scores.

Figure 8.6

Just as choosing a good number of `bins` is important for a good histogram, choosing a good `gridsize` is crucial for an informative hexbin plot. A large `gridsize` creates many small bins and a small `gridsize` creates fewer, larger bins.

#### NOTE

Since Hexbins are based on frequencies, they are prone to being misleading if the dataset is not understood well. For example, when plotting information that deals with geographic position, increases in frequency may be results in higher populations rather than the actual information being plotted.

See <http://pandas.pydata.org/pandas-docs/stable/visualization.html> for more types of plots available in Pandas and further examples.

**Problem 3.** Using distributional visualizations, display the following distributions:

1. The distribution of accidents across the United States
2. The distribution of age in all accidents

### 3. The distribution of fatalities by the hour which the accident occurs

## Principles of Good Data Visualization

Data visualization is a powerful tool for analysis and communication. When writing a paper or report, the author must make many decisions about how to use graphics effectively to convey useful information to the reader. Here we will go over a simple process for making deliberate, effective, and efficient design decisions.

### Attention to Detail

Consider the plot in Figure 8.7. What does it depict? It is a scatter plot of positively correlated data of some kind, with `temp`—likely temperature—on the  $x$  axis and `cons` on the  $y$  axis. However, the picture is not really communicating anything about the dataset. It has not specified the units for the  $x$  or the  $y$  axis, gives no understanding of what `cons` is. There is no title and the source of the data is unknown.

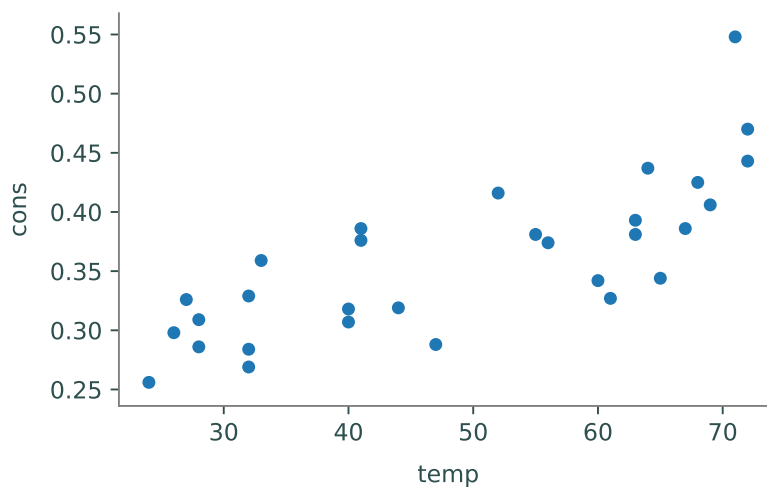


Figure 8.7: Non-specific data.

### Labels and Citations

In a homework or lab setting, we sometimes (mistakenly) think that it is acceptable to leave off appropriate labels, legends, titles, and sourcing. In a published report or presentation, this kind of carelessness is confusing at best and, when the source is not included, even plagiaristic. Data needs to be explained in a useful manner that includes all of the vital information.

Consider again Figure 8.7. This figure comes from the `Icecream` dataset within the `pydataset` package, which we store here in a dataframe and then plot:

```
>>> from pydataset import data
>>> icecream = data("Icecream")
>>> icecream.plot(kind="scatter", x="temp", y="cons")
```

This code produces the rather substandard plot in Figure 8.7. However, the method `data()` from the `pydataset` module has a boolean parameter `show_doc`, which displays information about the data being accessed.. Using `data('Icecream', show_doc=True)` we find the following information:

1. The dataset details ice cream consumption via four-weekly observations from March 1951 to July 1953 in the United States.
2. `cons` corresponds to “consumption of ice cream per head” and is measured in pints.
3. `temp` corresponds to temperature, degrees Fahrenheit.
4. The listed source is: “Hildreth, C. and J. Lu (1960) *Demand relations with autocorrelated disturbances*, Technical Bulletin No 2765, Michigan State University.”

This information gives important details that can be used in the following code. As seen in previous examples, pandas automatically generates legends when appropriate. Pandas also automatically labels the  $x$  and  $y$  axes, however our data frame column titles may be insufficient. Appropriate titles for the  $x$  and  $y$  axes must also list appropriate units. For example, the  $y$  axis should specify that the consumption is in units of *pints per head*, in place of the ambiguous label `cons`.

```
>>> icecream = data("Icecream")
# Set title via the title keyword argument
>>> icecream.plot(kind="scatter", x="temp", y="cons", title="Ice Cream ↵
Consumption in the U.S., 1951-1953",)
# Override pandas automatic labelling using xlabel and ylabel
>>> plt.xlabel("Temp (Fahrenheit)")
>>> plt.ylabel("Consumption per head (pints)")
```

To arbitrarily add the necessary text to the figure, use either `plt.annotate()` or `plt.text()`. Alternatively, add text immediately below wherever the figure is displayed.

```
>>> plt.text(20, .1, r"Source: Hildreth, C. and J. Lu (1960) \emph{Demand"
...      "relations with autocorrelated disturbances}\nTechnical Bulletin No"
...      "2765, Michigan State University.", fontsize=7)
```

Both of these methods are imperfect, but can normally be just as easily replaced by a caption attached to the figure in your presentation or document setting. Again, we reiterate how important it is that you source any data you use. Failing to do so is plagiarism.

Finally, we have a clear and demonstrative graphic in Figure 8.8.

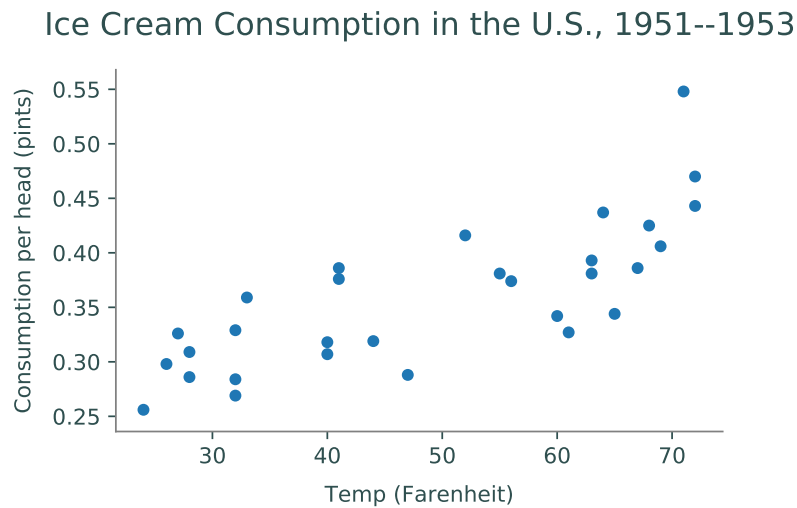


Figure 8.8: Source: Hildreth, C. and J. Lu (1960) *Demand relations with autocorrelated disturbances*, Technical Bulletin No 2765, Michigan State University.

### ACHTUNG!

Visualizing data can inherit many biases of the visualizer and as a result can be intentionally misleading. Examples of this include, but are not limited to, visualizing subsets of data that do not represent the whole of the data and having purposely misconstrued axes. Every data visualizer has the responsibility to avoid including biases in their visualizations to ensure data is being represented informatively and accurately.

**Problem 4.** The `pydataset` module contains numerous data sets, each stored as a pandas `DataFrame`. Read data on road accident deaths in the United States using the data set `road` in the `pydataset` module. Using this data, answer the following questions:

1. Does higher population density indicate higher fuel consumption?
2. Does a correlation exist between number of drivers and number of deaths in a state?
3. Does the temperature of a state in January affect its population density?

Support each claim with a visualization and make sure the visualizations are clearly labeled and easy to understand. Include any citations provided by the data set.

(Hint: Consider how outliers may be skewing the data.)

**Problem 5.** The file `new_york_crime_clean.csv` contains data on New York City crimes and felonies from 2000-2017 taken from <https://www1.nyc.gov/site/nypd/stats/crime-statistics/historical.page>. Use this data to answer the following questions and give visualizations supporting each claim:

1. What are the most common and least common crimes?
2. Does a trend exist between robbery and drug felonies? If so, what is the trend?
3. Which crimes have had the largest distributions?

Use a different style of visualization for each question. Make sure that each plot is well labeled and contains citations.

(Hint: Use `plt.axis('scaled')` or `plt.axis('square')` to make sure x- and y-axis are the same.)



# 9

## Pandas 3: Grouping

**Lab Objective:** *Many data sets contain categorical values that naturally sort the data into groups. Analyzing and comparing such groups is an important part of data analysis. In this lab we explore pandas tools for grouping data and presenting tabular data more compactly, primarily through groupby and pivot tables.*

### Groupby

The file `mammal_sleep.csv`<sup>1</sup> contains data on the sleep cycles of different mammals, classified by order, genus, species, and diet (carnivore, herbivore, omnivore, or insectivore). The `"sleep_total"` column gives the total number of hours that each animal sleeps (on average) every 24 hours. To get an idea of how many animals sleep for how long, we start off with a histogram of the `"sleep_total"` column.

```
>>> import pandas as pd
>>> from matplotlib import pyplot as plt

# Read in the data and print a few random entries.
>>> msleep = pd.read_csv("mammal_sleep.csv")
>>> msleep.sample(5)
   name   genus  vore   order  sleep_total  sleep_rem  sleep_cycle
51  Jaguar  Panthera  carn   Carnivora      10.4         NaN         NaN
77  Tenrec   Tenrec   omni  Afrosoricida      15.6          2.3         NaN
10   Goat    Capri   herbi  Artiodactyla       5.3          0.6         NaN
80   Genet   Genetta  carn   Carnivora       6.3          1.3         NaN
33   Human    Homo   omni    Primates       8.0          1.9          1.5

# Plot the distribution of the sleep_total variable.
>>> msleep.plot(kind="hist", y="sleep_total", title="Mammalian Sleep Data")
>>> plt.xlabel("Hours")
```

<sup>1</sup>Proceedings of the National Academy of Sciences, 104 (3):1051–1056, 2007. Updates from V. M. Savage and G. B. West, with additional variables supplemented by Wikipedia. Available in `pydataset` (with a few more columns) under the key `"msleep"`.

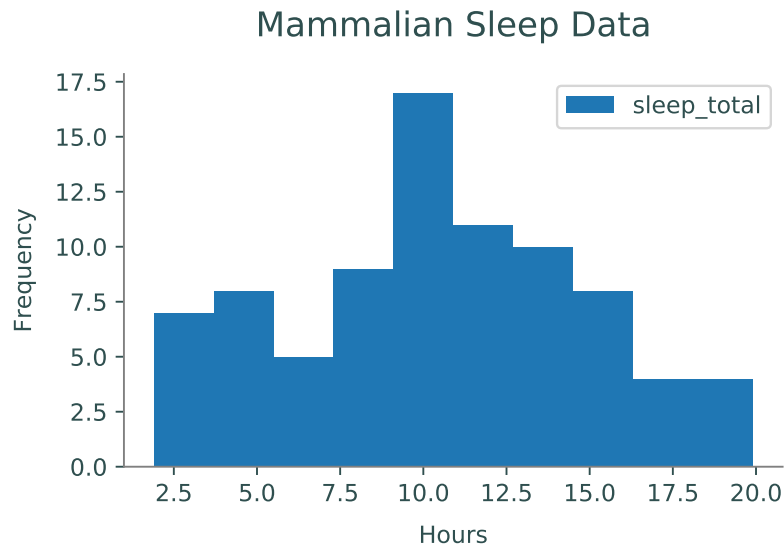


Figure 9.1: "sleep\_total" frequencies from the mammalian sleep data set.

While this visualization is a good start, it doesn't provide any information about how different kinds of animals have different sleeping habits. How long do carnivores sleep compared to herbivores? Do mammals of the same genus have similar sleep patterns?

A powerful tool for answering these kinds of questions is the `groupby()` method of the pandas `DataFrame` class, which partitions the original `DataFrame` into groups based on the values in one or more columns. The `groupby()` method does **not** return a new `DataFrame`; it returns a pandas `GroupBy` object, an interface for analyzing the original `DataFrame` by groups.

For example, the columns "genus", "vore", and "order" in the mammal sleep data all have a discrete number of categorical values that could be used to group the data. Since the "vore" column has only a few unique values, we start by grouping the animals by diet.

```
# List all of the unique values in the 'vore' column.
>>> set(msleep["vore"])
{nan, 'herbi', 'omni', 'carni', 'insecti'}

# Group the data by the 'vore' column.
>>> vores = msleep.groupby("vore")
>>> list(vores.groups)
['carni', 'herbi', 'insecti', 'omni']      # NaN values for vore were dropped.

# Get a single group and sample a few rows. Note vore='carni' in each entry.
>>> vores.get_group("carni").sample(5)
```

	name	genus	vore	order	sleep_total	sleep_rem	sleep_cycle
80	Genet	Genetta	carni	Carnivora	6.3	1.3	NaN
50	Tiger	Panthera	carni	Carnivora	15.8	NaN	NaN
8	Dog	Canis	carni	Carnivora	10.1	2.9	0.333
0	Cheetah	Acinonyx	carni	Carnivora	12.1	NaN	NaN
82	Red fox	Vulpes	carni	Carnivora	9.8	2.4	0.350



For starters, `groupby()` is useful for filtering a `DataFrame` by column values: the command `df.groupby(col).get_group(value)` returns the rows of `df` where the entry of the `col` column is `value`. The real advantage of `groupby()`, however, is how easy it makes it to compare groups of data. Standard `DataFrame` methods like `describe()`, `mean()`, `std()`, `min()`, and `max()` all work on `GroupBy` objects to produce a new data frame that describes the statistics of each group.

```
# Get averages of the numerical columns for each group.
>>> vores.mean()
           sleep_total  sleep_rem  sleep_cycle
vore
carni           10.379         2.290         0.373
herbi           9.509         1.367         0.418
insecti        14.940         3.525         0.161
omni           10.925         1.956         0.592

# Get more detailed statistics for 'sleep_total' by group.
>>> vores["sleep_total"].describe()
           count      mean      std  min   25%   50%   75%   max
vore
carni         19.0   10.379   4.669   2.7   6.25  10.4  13.000  19.4
herbi         32.0    9.509   4.879   1.9   4.30  10.3  14.225  16.6
insecti        5.0   14.940   5.921   8.4   8.60  18.1  19.700  19.9
omni          20.0   10.925   2.949   8.0   9.10   9.9  10.925  18.0
```

Multiple columns can be used simultaneously for grouping. In this case, the `get_group()` method of the `GroupBy` object requires a tuple specifying the values for each of the grouping columns.

```
>>> msleep_small = msleep.drop(["sleep_rem", "sleep_cycle"], axis=1)
>>> vores_orders = msleep_small.groupby(["vore", "order"])
>>> vores_orders.get_group(("carni", "Cetacea"))
           name           genus  vore  order  sleep_total
30      Pilot whale  Globicephalus  carn  Cetacea         2.7
59   Common porpoise      Phocoena  carn  Cetacea         5.6
79  Bottle-nosed dolphin      Tursiops  carn  Cetacea         5.2
```

## Visualizing Groups

There are a few ways that `groupby()` or similar techniques can simplify the process of visualizing groups of data. First of all, `groupby()` makes it easy to visualize one group at a time. The following visualization improve on Figure 9.1 by grouping mammals by their diets.

```
# Plot histograms of 'sleep_total' for two separate groups.
>>> vores.get_group("carni").plot(kind="hist", y="sleep_total", legend="False",
                                title="Carnivore Sleep Data")

>>> plt.xlabel("Hours")
>>> vores.get_group("herbi").plot(kind="hist", y="sleep_total", legend="False",
                                title="Herbivore Sleep Data")

>>> plt.xlabel("Hours")
```

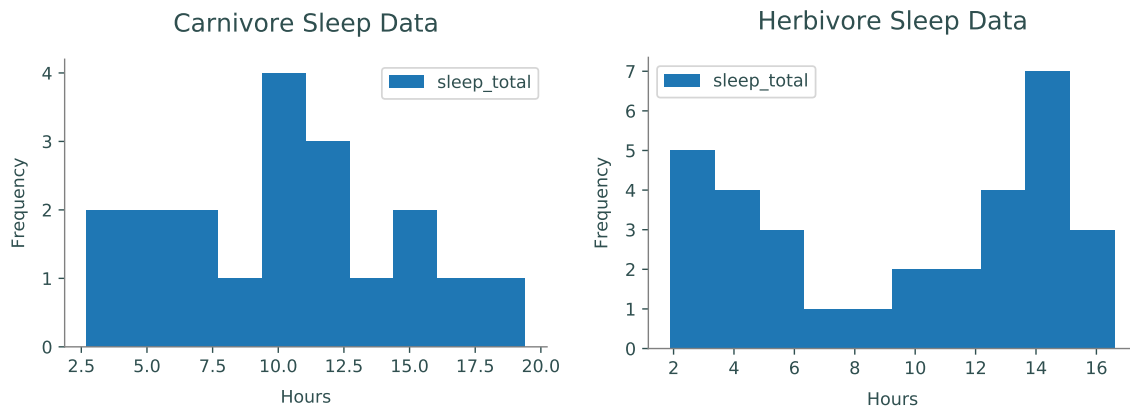
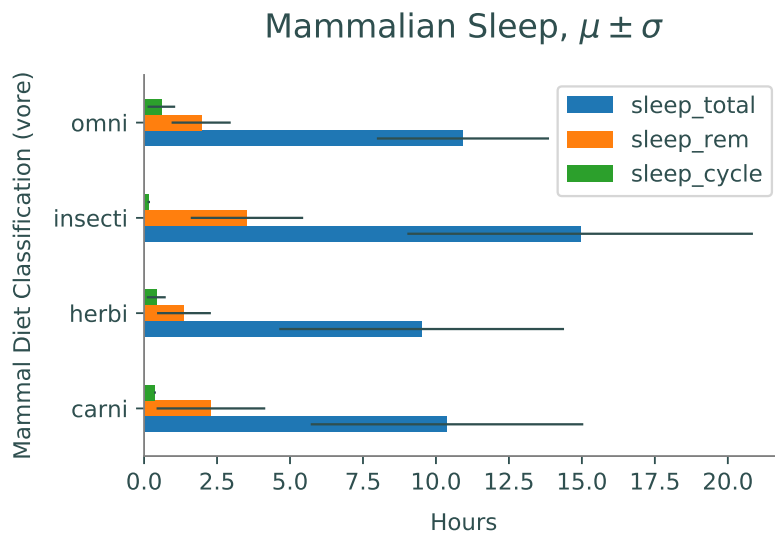


Figure 9.2: "sleep\_total" histograms for two groups in the mammalian sleep data set.

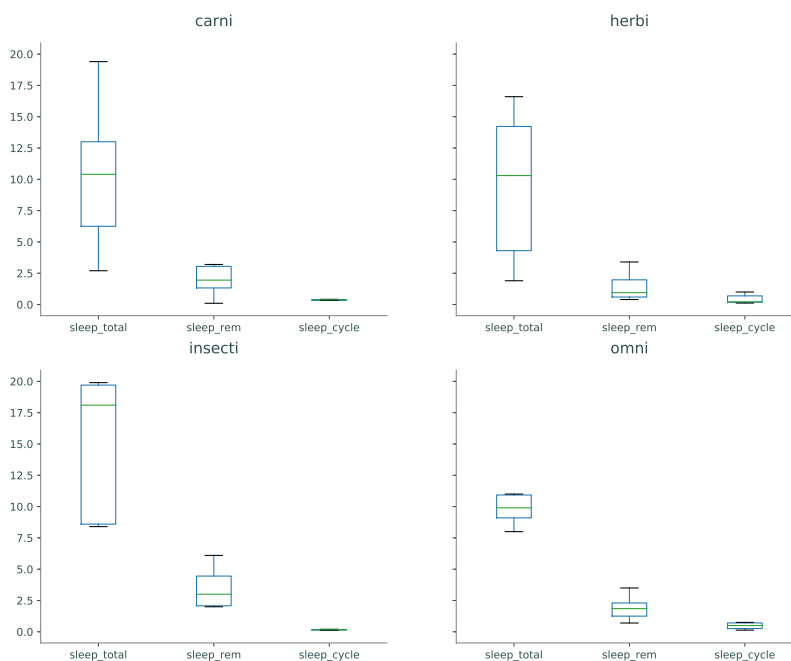
The statistical summaries from the `GroupBy` object's `mean()`, `std()`, or `describe()` methods also lend themselves well to certain visualizations for comparing groups.

```
>>> vores[["sleep_total", "sleep_rem", "sleep_cycle"]].mean().plot(kind="barh",
                        xerr=vores.std(), title=r"Mammalian Sleep,  $\mu \pm \sigma$ ")
>>> plt.xlabel("Hours")
>>> plt.ylabel("Mammal Diet Classification (vore)")
```



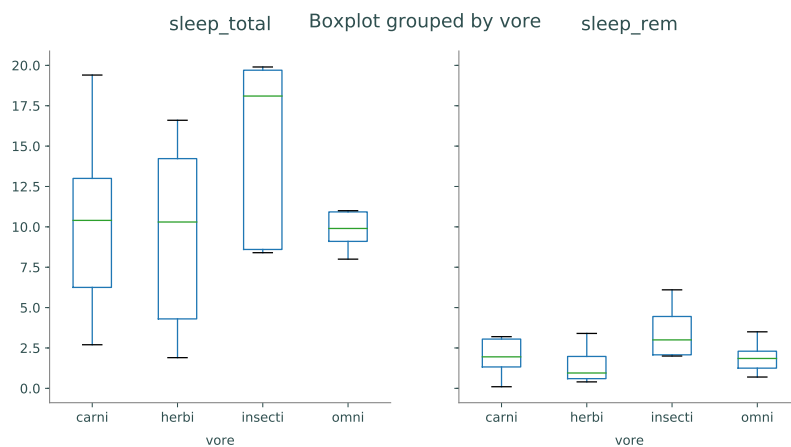
Box plots are well suited for comparing similar distributions. The `boxplot()` method of the `GroupBy` class creates one subplot **per group**, plotting each of the columns as a box plot.

```
# Use GroupBy.boxplot() to generate one box plot per group.
>>> vores.boxplot(grid=False)
>>> plt.tight_layout()
```



Alternatively, the `boxplot()` method of the `DataFrame` class creates one subplot **per column**, plotting each of the columns as a box plot. Specify the `by` keyword to group the data appropriately.

```
# Use DataFrame.boxplot() to generate one box plot per column.
>>> msleep.boxplot(["sleep_total", "sleep_rem"], by="vore", grid=False)
```



Like `groupby()`, the `by` argument can be a single column label or a list of column labels. Similar methods exist for creating histograms (`GroupBy.hist()` and `DataFrame.hist()` with `by` keyword), but generally box plots are better for comparing multiple distributions.

**Problem 1.** Examine the following data sets from `pydataset` and answer the corresponding questions. Use visualizations to support your conclusions.

- **"iris"**, measurements of various species of iris flowers.
  1. Which species is easiest to distinguish from the others? How?
  2. Given iris data without a species label, what strategies could you use to identify the flower's species?
- **"poisons"**, experimental results of three different poisons and four different treatments.
  1. In general, which poison is most deadly? Which treatment is most effective?
  2. If you were poisoned, how would you choose the treatment if you did not know which poison it was? What if you did know which poison it was?  
(Hint: group the data by poison, then group each subset by treatment.)
- **"diamonds"**, prices and characteristics of almost 54,000 round-cut diamonds.
  1. How does the color and cut of a diamond affect its price?
  2. Of the diamonds with color **"H"**, those with a **"Fair"** cut sell, on average, for a **higher** price than those with an **"Ideal"** (superior) cut. What other factors could explain this unintuitive statistic?

## Pivot Tables

One of the downsides of `groupby()` is that a typical `GroupBy` object has too much information to display coherently. A *pivot table* intelligently summarizes the results of a `groupby()` operation by aggregating the data in a specified way. The standard tool for making a pivot table is the `pivot_table()` method of the `DataFrame` class. As an example, consider the **"HairEyeColor"** data set from `pydataset`.

```
>>> from pydataset import data
>>> hec = data("HairEyeColor")                # Load and preview the data.
>>> hec.sample(5)
   Hair  Eye  Sex  Freq
3   Red  Brown  Male   10
1  Black  Brown  Male   32
14 Brown  Green  Male   15
31   Red  Green  Female   7
21  Black  Blue  Female   9

>>> for col in ["Hair", "Eye", "Sex"]:        # Get unique values per column.
...     print("{}: {}".format(col, " ".join(set(str(x) for x in hec[col]))))
...
Hair: Brown, Black, Blond, Red
Eye: Brown, Blue, Hazel, Green
Sex: Male, Female
```

There are several ways to group this data with `groupby()`. However, since there is only one entry per unique hair-eye-sex combination, the data can be completely presented in a pivot table.

```
>>> hec.pivot_table(values="Freq", index=["Hair", "Eye"], columns="Sex")
```

Sex		Female	Male
Hair	Eye		
Black	Blue	9	11
	Brown	36	32
	Green	2	3
	Hazel	5	10
Blond	Blue	64	30
	Brown	4	3
	Green	8	8
	Hazel	5	5
Brown	Blue	34	50
	Brown	66	53
	Green	14	15
	Hazel	29	25
Red	Blue	7	10
	Brown	16	10
	Green	7	7
	Hazel	7	7

Listing the data in this way makes it easy to locate data and compare the female and male groups. For example, it is easy to see that brown hair is more common than red hair and that about twice as many females have blond hair and blue eyes than males.

Unlike `"HairEyeColor"`, many data sets have more than one entry in the data for each grouping (for example, if there were two or more rows in the original data for females with blond hair and blue eyes). To construct a pivot table, data of similar groups must be *aggregated* together in some way. By default entries are aggregated by averaging the non-null values. Other options include taking the min, max, standard deviation, or just counting the number of occurrences.

As an example, consider again the Titanic data set found in `titanic.csv`<sup>2</sup>. For this analysis, take only the `"Survived"`, `"Pclass"`, `"Sex"`, `"Age"`, `"Fare"`, and `"Embarked"` columns, replace null age values with the average age, then drop any rows that are missing data. To begin, we examine the average survival rate grouped by sex and passenger class.

```
>>> titanic = pd.read_csv("titanic.csv")
>>> titanic = titanic[["Survived", "Pclass", "Sex", "Age", "Fare", "Embarked"]]
>>> titanic["Age"].fillna(titanic["Age"].mean(), inplace=True)
>>> titanic.dropna(inplace=True)

>>> titanic.pivot_table(values="Survived", index="Sex", columns="Pclass")
```

Pclass	1.0	2.0	3.0
Sex			
female	0.965	0.887	0.491
male	0.341	0.146	0.152

<sup>2</sup>There is a `"Titanic"` data set in `pydataset`, but it does not contain as much information as the data in `titanic.csv`.

## NOTE

The `pivot_table()` method is just a convenient way of performing a potentially complicated `groupby()` operation with aggregation and some reshaping. For example, the following code is equivalent to the previous example.

```
>>> titanic.groupby(["Sex", "Pclass"])["Survived"].mean().unstack()
Pclass    1.0    2.0    3.0
Sex
female  0.965  0.887  0.491
male    0.341  0.146  0.152
```

The `stack()`, `unstack()`, and `pivot()` methods provide more advanced shaping options.

Among other things, this pivot table clearly shows how much more likely females were to survive than males. To see how many entries fall into each category, or how many survived in each category, aggregate by counting or summing instead of taking the mean.

```
# See how many entries are in each category.
>>> titanic.pivot_table(values="Survived", index="Sex", columns="Pclass",
...                      aggfunc="count")
Pclass  1.0  2.0  3.0
Sex
female  144  106  216
male    179  171  493

# See how many people from each category survived.
>>> titanic.pivot_table(values="Survived", index="Sex", columns="Pclass",
...                      aggfunc="sum")
Pclass    1.0    2.0    3.0
Sex
female  137.0  94.0  106.0
male     61.0  25.0   75.0
```

## Discretizing Continuous Data

So far we have examined survival rates based on sex and passenger class. Another factor that could have played into survival is age. Were male children as likely to die as females in general? We can investigate this question by *multi-indexing*, or pivoting on more than just two variables, by adding in another index.

In the original dataset, the `"Age"` column has a floating point value for the age of each passenger. If we just added `"Age"` as another pivot, then the table would create a new row for **each** age present. Instead, we partition the `"Age"` column into intervals with `pd.cut()`, thus creating a categorical that can be used for grouping.

```
# pd.cut() maps continuous entries to discrete intervals.
>>> pd.cut([6, 1, 2, 3, 4, 5, 6, 7], [0, 4, 8])
[(0, 4], (0, 4], (0, 4], (0, 4], (4, 8], (4, 8], (4, 8], (0, 4]]
Categories (2, interval[int64]): [(0, 4] < (4, 8]]

# Partition the passengers into 3 categories based on age.
>>> age = pd.cut(titanic['Age'], [0, 12, 18, 80])

>>> titanic.pivot_table(values="Survived", index=["Sex", age],
                        columns="Pclass", aggfunc="mean")
Pclass      1.0      2.0      3.0
Sex  Age
female (0, 12]  0.000  1.000  0.467
        (12, 18]  1.000  0.875  0.607
        (18, 80]  0.969  0.871  0.475
male   (0, 12]  1.000  1.000  0.343
        (12, 18]  0.500  0.000  0.081
        (18, 80]  0.322  0.093  0.143
```

From this table, it appears that male children (ages 0 to 12) in the 1st and 2nd class were very likely to survive, whereas those in 3rd class were much less likely to. This clarifies the claim that males were less likely to survive than females. However, there are a few oddities in this table: zero percent of the female children in 1st class survived, and zero percent of teenage males in second class survived. To further investigate, count the number of entries in each group.

```
>>> titanic.pivot_table(values="Survived", index=["Sex", age],
                        columns="Pclass", aggfunc="count")
Pclass      1.0      2.0      3.0
Sex  Age
female (0, 12]      1     13     30
        (12, 18]     12      8     28
        (18, 80]    129     85    158
male   (0, 12]       4     11     35
        (12, 18]       4     10     37
        (18, 80]    171    150    420
```

This table shows that there was only 1 female child in first class and only 10 male teenagers in second class, which sheds light on the previous table.

### ACHTUNG!

The previous pivot table brings up an important point about partitioning datasets. The Titanic dataset includes data for about 1300 passengers, which is a somewhat reasonable sample size, but half of the groupings include less than 30 entries, which is **not** a healthy sample size for statistical analysis. Always carefully question the numbers from pivot tables before making any conclusions.

Pandas also supports multi-indexing on the columns. As an example, consider the price of a passenger tickets. This is another continuous feature that can be discretized with `pd.cut()`. Instead, we use `pd.qcut()` to split the prices into 2 equal quantiles. Some of the resulting groups are empty; to improve readability, specify `fill_value` as the empty string or a dash.

```
# pd.qcut() partitions entries into equally populated intervals.
>>> pd.qcut([1, 2, 5, 6, 8, 3], 2)
[(0.999, 4.0], (0.999, 4.0], (4.0, 8.0], (4.0, 8.0], (4.0, 8.0], (0.999, 4.0]]
Categories (2, interval[float64]): [(0.999, 4.0] < (4.0, 8.0]]

# Cut the ticket price into two intervals (cheap vs expensive).
>>> fare = pd.qcut(titanic["Fare"], 2)
>>> titanic.pivot_table(values="Survived",
                        index=["Sex", age], columns=[fare, "Pclass"],
                        aggfunc="count", fill_value='-')
Fare          (-0.001, 14.454]      (14.454, 512.329]
Pclass
Sex  Age
female (0, 12]          -   -   7              1   13   23
      (12, 18]          -   4   23             12   4   5
      (18, 80]          -  31  101            129  54  57
male   (0, 12]          -   -   8              4   11  27
      (12, 18]          -   5   26             4   5   11
      (18, 80]          8  94  350            163  56  70
```

Not surprisingly, most of the cheap tickets went to passengers in 3rd class.

**Problem 2.** Suppose that someone claims that the city from which a passenger embarked had a strong influence on the passenger's survival rate. Investigate this claim.

1. Check the survival rates of the passengers based on where they embarked from (given in the `"Embarked"` column).
2. Create a pivot table to examine survival rates based on both place of embarkment and gender.
3. What do these tables suggest to you about the significance of where people embarked in influencing their survival rate? Examine the context of the problem, and explain what you think this really means.
4. Investigate the claim further with at least two more pivot tables, exploring other criteria (e.g., class, age, etc.). Carefully explain your conclusions.



**Problem 3.** Examine the following data sets from `pydataset` and answer the corresponding questions. Use visualizations and/or pivot tables as appropriate to support your conclusions.

- **"npk"**, an experiment on the effects of nitrogen (N), phosphate (P), and potassium (K) on the growth of peas.
  1. Which element is most effective in general for simulating growth? Which is the least effective?
  2. What combination of N, P, and K is optimal? What combination is the worst?
- **"swiss"**, standardized fertility measures and socio-economic indicators for French-speaking provinces of Switzerland at about 1888.
  1. What is the relationship in the data between fertility rates and infant mortality?
  2. How are provinces that are predominantly Catholic different from non-Catholic provinces, if at all?
  3. What factors in the data are the most important for predicting fertility?
- Examine a data set of your choice. Formulate simple questions about the data and hypothesize the answers to those questions. Demonstrate the correctness of incorrectness of each hypothesis. Explain your conclusions.



# 10 Pandas 4: Time Series

**Lab Objective:** *Many real world data sets—stock market measurements, ocean tide levels, website traffic, seismograph data, audio signals, fluid simulations, quarterly dividends, and so on—are time series, meaning they come with time-based labels. There is no universal format for such labels, and indexing by time is often difficult with raw data. Fortunately, pandas has tools for cleaning and analyzing time series. In this lab we use pandas to clean and manipulate time-stamped data and introduce some basic tools for time series analysis.*

## Working with Dates and Times

The `datetime` module in the standard library provides a few tools for representing and operating on dates and times. The `datetime.datetime` object represents a *time stamp*: a specific time of day on a certain day. Its constructor accepts a four-digit year, a month (starting at 1 for January), a day, and, optionally, an hour, minute, second, and microsecond. Each of these arguments must be an integer, with the hour ranging from 0 to 23 (military time).

```
>>> from datetime import datetime

# Represent November 18th, 1991, at 2:01 PM.
>>> bday = datetime(1991, 11, 18, 14, 1)
>>> print(bday)
1991-11-18 14:01:00

# Find the number of days between 11/18/1991 and 11/9/2017.
>>> dt = datetime(2017, 11, 9) - bday
>>> dt.days
9487
```

The `datetime.datetime` object has a parser method, `strptime()`, that converts a string into a new `datetime.datetime` object. The parser is flexible because the user must specify the format that the dates are in. For example, if the dates are in the format "**Month/Day//Year::Hour**", specify `format="%m/%d//%Y::%H"` to parse the string appropriately. See Table 10.1 for formatting options.

Pattern	Description
%Y	4-digit year
%y	2-digit year
%m	1- or 2-digit month
%d	1- or 2-digit day
%H	Hour (24-hour)
%I	Hour (12-hour)
%M	2-digit minute
%S	2-digit second

Table 10.1: Formats recognized by `datetime.strptime()`

```
>>> print(datetime.strptime("1991-11-18 / 14:01", "%Y-%m-%d / %H:%M"),
...       datetime.strptime("1/22/1996", "%m/%d/%Y"),
...       datetime.strptime("19-8, 1998", "%d-%m, %Y"), sep='\n')
1991-11-18 14:01:00      # The date formats are now standardized.
1996-01-22 00:00:00      # If no hour/minute/seconds data is given,
1998-08-19 00:00:00      # the default is midnight.
```

## Converting Dates to an Index

The `TimeStamp` class is the pandas equivalent to a `datetime.datetime` object. A pandas index composed of `TimeStamp` objects is a `DatetimeIndex`, and a `Series` or `DataFrame` with such an index is called a *time series*. The function `pd.to_datetime()` converts a collection of dates in a parsable format to a `DatetimeIndex`. The format of the dates is inferred if possible, but it can be specified explicitly with the same syntax as `datetime.strptime()`.

```
>>> import pandas as pd

# Convert some dates (as strings) into a DatetimeIndex.
>>> dates = ["2010-1-1", "2010-2-1", "2012-1-1", "2012-1-2"]
>>> pd.to_datetime(dates)
DatetimeIndex(['2010-01-01', '2010-02-01', '2012-01-01', '2012-01-02'],
              dtype='datetime64[ns]', freq=None)

# Create a time series, specifying the format for the dates.
>>> dates = ["1/1, 2010", "1/2, 2010", "1/1, 2012", "1/2, 2012"]
>>> date_index = pd.to_datetime(dates, format="%m/%d, %Y")
>>> pd.Series([x**2 for x in range(4)], index=date_index)
2010-01-01    0
2010-01-02    1
2012-01-01    4
2012-01-02    9
dtype: int64
```

**Problem 1.** The file `DJIA.csv` contains daily closing values of the Dow Jones Industrial Average from 2006–2016. Read the data into a `Series` or `DataFrame` with a `DatetimeIndex` as the index. Drop rows with missing values, cast the **"VALUES"** column to floats, then plot the data. (Hint: Use `lw=.5` to make the line thin enough for the data.)

## Generating Time-based Indices

Some time series datasets come without explicit labels, but still have instructions for deriving timestamps. For example, a list of bank account balances might have records from the beginning of every month, or heart rate readings could be recorded by an app every 10 minutes. Use `pd.date_range()` to generate a `DatetimeIndex` where the timestamps are equally spaced. The function is analogous to `np.arange()` and has the following parameters.

Parameter	Description
<code>start</code>	Starting date
<code>end</code>	End date
<code>periods</code>	Number of dates to include
<code>freq</code>	Amount of time between consecutive dates
<code>normalize</code>	Whether or not to trim the time to midnight

Table 10.2: Parameters for `pd.date_range()`.

Exactly two of the parameters `start`, `end`, and `periods` must be specified to generate a range of dates. The `freq` parameter accepts a variety of string representations, referred to as *offset aliases*. See Table 10.3 for a sampling of some of the options. For a complete list of the options, see <http://pandas.pydata.org/pandas-docs/stable/timeseries.html#offset-aliases>.

Parameter	Description
<b>"D"</b>	calendar daily (default)
<b>"B"</b>	business daily
<b>"H"</b>	hourly
<b>"T"</b>	minutely
<b>"S"</b>	secondly
<b>"MS"</b>	first day of the month
<b>"BMS"</b>	first weekday of the month
<b>"W-MON"</b>	every Monday
<b>"WOM-3FRI"</b>	every 3rd Friday of the month

Table 10.3: Options for the `freq` parameter to `pd.date_range()`.

```
# 5 consecutive days starting with September 28, 2016.
>>> pd.date_range(start='9/28/2016 16:00', periods=5)
DatetimeIndex(['2016-09-28 16:00:00', '2016-09-29 16:00:00',
               '2016-09-30 16:00:00', '2016-10-01 16:00:00',
               '2016-10-02 16:00:00'],
              dtype='datetime64[ns]', freq='D')
```

```

dtype='datetime64[ns]', freq='D')

# The first weekday of every other month in 2016.
>>> pd.date_range(start='1/1/2016', end='1/1/2017', freq="2BMS" )
DatetimeIndex(['2016-01-01', '2016-03-01', '2016-05-02', '2016-07-01',
              '2016-09-01', '2016-11-01'],
              dtype='datetime64[ns]', freq='2BMS')

# 10 minute intervals between 4:00 PM and 4:30 PM on September 9, 2016.
>>> pd.date_range(start='9/28/2016 16:00',
                  end='9/28/2016 16:30', freq="10T")
DatetimeIndex(['2016-09-28 16:00:00', '2016-09-28 16:10:00',
              '2016-09-28 16:20:00', '2016-09-28 16:30:00'],
              dtype='datetime64[ns]', freq='10T')

```

The `freq` parameter also supports more flexible string representations.

```

>>> pd.date_range(start='9/28/2016 16:30', periods=5, freq="2h30min")
DatetimeIndex(['2016-09-28 16:30:00', '2016-09-28 19:00:00',
              '2016-09-28 21:30:00', '2016-09-29 00:00:00',
              '2016-09-29 02:30:00'],
              dtype='datetime64[ns]', freq='150T')

```

**Problem 2.** The file `paychecks.csv` contains values of an hourly employee's last 93 paychecks. He started working March 13, 2008. This company hands out paychecks on the first and third Fridays of the month.

Read in the data, using `pd.date_range()` to generate the `DatetimeIndex`. Plot the data. (Hint: use the `union()` method of `DatetimeIndex` class.)

## Periods

The pandas `Timestamp` object represents a precise moment in time on a given day. Some data, however, is recorded over a time interval, and it wouldn't make sense to place an exact timestamp on any of the measurements. For example, a record of the number of steps walked in a day, box office earnings per week, quarterly earnings, and so on. This kind of data is better represented with the pandas `Period` object and the corresponding `PeriodIndex`.

The constructor of the `Period` accepts a `value` and a `freq`. The `value` parameter indicates the label for a given `Period`. This label is tied to the **end** of the defined `Period`. The `freq` indicates the length of the `Period` and also (in some cases) indicates the offset of the `Period`. The `freq` parameter accepts the majority, but not all, of frequencies listed in Table 10.3.

```

# The default value for 'freq' is "M" for months.
>>> p1 = pd.Period("2016-10")
>>> p1.start_time          # The start and end times of the period
Timestamp('2016-10-01 00:00:00') # are recorded as Timestamps.

```

```
>>> p1.end_time
Timestamp('2016-10-31 23:59:59.999999999')

# Represent the annual period ending in December that includes 10/03/2016.
>>> p2 = pd.Period("2016-10-03", freq="A-DEC")
>>> p2.start_time
Timestamp('2007-01-01 00:00:00')
>>> p2.end_time
Timestamp('2007-12-31 23:59:59.999999999')

# Get the weekly period ending on a Saturday that includes 10/03/2016.
>>> print(pd.Period("2016-10-03", freq="W-SAT"))
2016-10-02/2016-10-08
```

Like the `pd.date_range()` method, the `pd.period_range()` method is useful for generating a `PeriodIndex` for unindexed data. The syntax is essentially identical to that of `pd.date_range()`. When using `pd.period_range()`, remember that the `freq` parameter marks the end of the period.

```
# Represent quarters from 2008 to 2010, with Q4 ending in December.
>>> pd.period_range(start="2008", end="2010-12", freq="Q-DEC")
PeriodIndex(['2008Q1', '2008Q2', '2008Q3', '2008Q4', '2009Q1', '2009Q2',
            '2009Q3', '2009Q4', '2010Q1', '2010Q2', '2010Q3', '2010Q4'],
            dtype='period[Q-DEC]', freq='Q-DEC')
```

After creating a `PeriodIndex`, the `freq` parameter can be changed via the `asfreq()` method.

```
# Get every three months form March 2010 to the start of 2011.
>>> p = pd.period_range("2010-03", "2011", freq="3M")
>>> p
PeriodIndex(['2010-03', '2010-06', '2010-09', '2010-12'],
            dtype='period[3M]', freq='3M')

# Change frequency to be quarterly.
>>> p.asfreq("Q-DEC")
PeriodIndex(['2010Q2', '2010Q3', '2010Q4', '2011Q1'],
            dtype='period[Q-DEC]', freq='Q-DEC')
```

Say you have created a `PeriodIndex`, but the bounds are not exactly where you expected they would be. You can actually shift `PeriodIndex` objects by adding or subtracting an integer,  $n$ . The resulting `PeriodIndex` will be shifted by  $n \times \text{freq}$ .

```
# Shift index by 1
>>> p -= 1
>>> p
PeriodIndex(['2010Q1', '2010Q2', '2010Q3', '2010Q4'],
            dtype='int64', freq='Q-DEC')
```

If for any reason you need to switch from periods to timestamps, pandas provides a very simple method to do so.

```
# Convert to timestamp (last day of each quarter)
>>> p = p.to_timestamp(how='end')
>>> p
DatetimeIndex(['2010-03-31', '2010-06-30', '2010-09-30', '2010-12-31'],
              dtype='datetime64[ns]', freq='Q-DEC')
```

Similarly, you can switch from timestamps to periods.

```
>>> p.to_period("Q-DEC")
PeriodIndex(['2010Q1', '2010Q2', '2010Q3', '2010Q4'],
           dtype='int64', freq='Q-DEC')
```

**Problem 3.** The file `finances.csv` contains a list of simulated quarterly earnings and expense totals from a fictional company. Load the data into a `Series` or `DataFrame` with a `PeriodIndex` with a quarterly frequency. Assume the fiscal year starts at the beginning of September and that the data begins in September 1978. Plot the data.

## Operations on Time Series

There are certain operations only available to `Series` and `DataFrames` that have a `DatetimeIndex`. A sampling of this functionality is described throughout the remainder of this lab.

### Slicing

Slicing is much more flexible in pandas for time series. We can slice by year, by month, or even use traditional slicing syntax to select a range of dates.

```
# Select all rows in a given year
>>> df["2010"]
           0          1
2010-01-01  0.566694  1.093125
2010-02-01 -0.219856  0.852917
2010-03-01  1.511347 -1.324036

# Select all rows in a given month of a given year
>>> df["2012-01"]
           0          1
2012-01-01  0.212141  0.859555
2012-01-02  1.483123 -0.520873
2012-01-03  1.436843  0.596143

# Select a range of dates using traditional slicing syntax
>>> df["2010-1-2":"2011-12-31"]
           0          1
2010-02-01 -0.219856  0.852917
```



2010-03-01	1.511347	-1.324036
2011-01-01	0.300766	0.934895

## Resampling

Imagine you have a dataset that does not have datapoints at a fixed frequency. For example, a dataset of website traffic would take on this form. Because the datapoints occur at irregular intervals, it may be more difficult to procure any meaningful insight. In situations like these, *resampling* your data is worth considering.

The two main forms of resampling are *downsampling* (aggregating data into fewer intervals) and *upsampling* (adding more intervals).

To downsample, use the `resample()` method of the `Series` or `DataFrame`. This method is similar to `groupby()` in that it groups different entries together, and requires some kind of aggregation to produce a new data set. The first parameter to `resample()` is an offset string from Table 10.3: `"D"` for daily, `"H"` for hourly, and so on.

```
>>> import numpy as np

# Get random data for every day from 2000 to 2010.
>>> dates = pd.date_range(start="2000-1-1", end='2009-12-31', freq='D')
>>> df = pd.Series(np.random.random(len(dates)), index=dates)
>>> df
2000-01-01    0.559
2000-01-02    0.874
2000-01-03    0.774
...
2009-12-29    0.837
2009-12-30    0.472
2009-12-31    0.211
Freq: D, Length: 3653, dtype: float64

# Group the data by year.
>>> years = df.resample("A")           # 'A' for 'annual'.
>>> years.agg(len)                   # Number of entries per year.
2000-12-31    366.0
2001-12-31    365.0
2002-12-31    365.0
...
2007-12-31    365.0
2008-12-31    366.0
2009-12-31    365.0
Freq: A-DEC, dtype: float64

>>> years.mean()                   # Average entry by year.
2000-12-31    0.491
2001-12-31    0.514
2002-12-31    0.484
...
```

```

2007-12-31    0.508
2008-12-31    0.521
2009-12-31    0.523
Freq: A-DEC, dtype: float64

# Group the data by month.
>>> months = df.resample("M")
>>> len(months.mean())           # 12 months x 10 years = 120 months.
120

```

**Problem 4.** The file `website_traffic.csv` contains records for different visits to a fictitious website. Read in the data, calculate the duration of each visit (in seconds), and convert the index to a `DatetimeIndex`. Use downsampling to calculate the average visit duration by minute, and the average visit duration by hour. Plot both results on the same graph.

## Elementary Time Series Analysis

### Shifting

`DataFrame` and `Series` objects have a `shift()` method that allows you to move data up or down relative to the index. When dealing with time series data, we can also shift the `DatetimeIndex` relative to a time offset.

```

>>> df = pd.DataFrame(dict(VALUE=np.random.rand(5)),
                          index=pd.date_range("2016-10-7", periods=5, freq='D'))
>>> df
              VALUE
2016-10-07  0.127895
2016-10-08  0.811226
2016-10-09  0.656711
2016-10-10  0.351431
2016-10-11  0.608767

>>> df.shift(1)
              VALUE
2016-10-07      NaN
2016-10-08  0.127895
2016-10-09  0.811226
2016-10-10  0.656711
2016-10-11  0.351431

>>> df.shift(-2)
              VALUE
2016-10-07  0.656711
2016-10-08  0.351431
2016-10-09  0.608767

```

```

2016-10-10      NaN
2016-10-11      NaN

>>> df.shift(14, freq="D")
      VALUE
2016-10-21  0.127895
2016-10-22  0.811226
2016-10-23  0.656711
2016-10-24  0.351431
2016-10-25  0.608767

```

Shifting data makes it easy to gather statistics about changes from one timestamp or period to the next.

```

# Find the changes from one period/timestamp to the next
>>> df - df.shift(1)      # Equivalent to df.diff().
      VALUE
2016-10-07      NaN
2016-10-08  0.683331
2016-10-09 -0.154516
2016-10-10 -0.305279
2016-10-11  0.257336

```

**Problem 5.** Compute the following information about the DJIA dataset from Problem 1.

- The single day with the largest gain.
- The single day with the largest loss.
- The month with the largest gain.
- The month with the largest loss.

For the monthly statistics, define the gain (or loss) to be the difference between the DJIA on the last and first days of the month.

## Rolling Functions and Exponentially-Weighted Moving Functions

Many time series are inherently noisy. To analyze general trends in data, we use *rolling functions* and *exponentially-weighted moving (EWM) functions*.

Rolling functions, or *moving window functions*, perform some kind of calculation on just a window of data. There are a few rolling functions that come standard with pandas.

### Rolling Functions (Moving Window Functions)

One of the most commonly used rolling functions is the *rolling average*, which takes the average value over a window of data.

```
# Generate a time series using random walk from a uniform distribution.
N = 10000
bias = 0.01
s = np.zeros(N)
s[1:] = np.random.uniform(low=-1, high=1, size=N-1) + bias
s = pd.Series(s.cumsum(),
              index=pd.date_range("2015-10-20", freq='H', periods=N))

# Plot the original data together with a rolling average.
ax1 = plt.subplot(121)
s.plot(color="gray", lw=.3, ax=ax1)
s.rolling(window=200).mean().plot(color='r', lw=1, ax=ax1)
ax1.legend(["Actual", "Rolling"], loc="lower right")
ax1.set_title("Rolling Average")
```

The function call `s.rolling(window=200)` creates a `pd.core.rolling.Window` object that can be aggregated with a function like `mean()`, `std()`, `var()`, `min()`, `max()`, and so on.

### Exponentially-Weighted Moving (EWM) Functions

Whereas a moving window function gives equal weight to the whole window, an *exponentially-weighted moving* function gives more weight to the most recent data points.

In the case of a *exponentially-weighted moving average* (EWMA), each data point is calculated as follows.

$$z_i = \alpha \bar{x}_i + (1 - \alpha) z_{i-1},$$

where  $z_i$  is the value of the EWMA at time  $i$ ,  $\bar{x}_i$  is the average for the  $i$ -th window, and  $\alpha$  is the decay factor that controls the importance of previous data points. Notice that  $\alpha = 1$  reduces to the rolling average.

More commonly, the decay is expressed as a function of the window size. In fact, the `span` for an EWMA is nearly analogous to `window` size for a rolling average.

Notice the syntax for EWM functions is very similar to that of rolling functions.

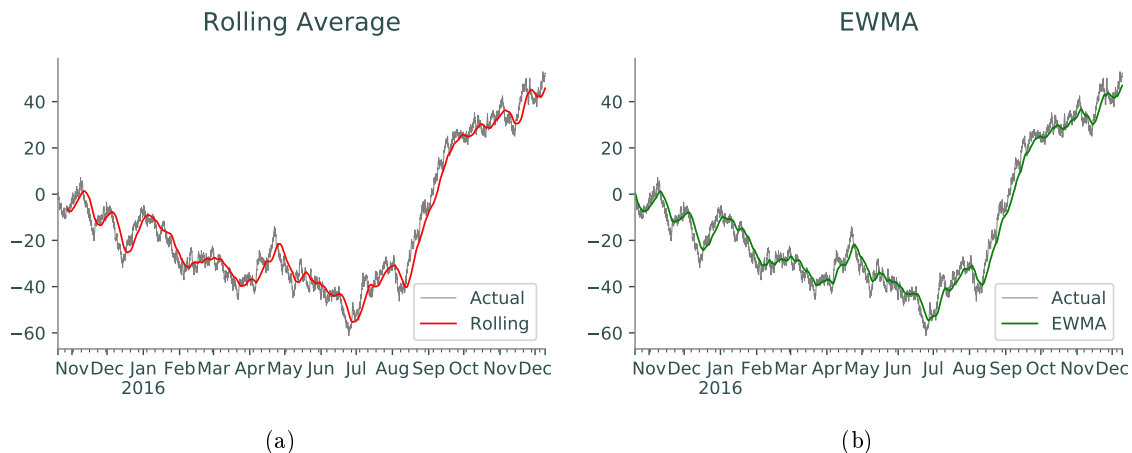


Figure 10.1: Rolling average and EWMA.

```
ax2 = plt.subplot(121)
s.plot(color="gray", lw=.3, ax=ax2)
s.ewm(span=200).mean().plot(color='g', lw=1, ax=ax2)
ax2.legend(["Actual", "EWMA"], loc="lower right")
ax2.set_title("EWMA")
```

**Problem 6.** Plot the following from the DJIA dataset with a window or span of 30, 120, and 365.

- The original data points.
- Rolling average.
- Exponential average.
- Minimum rolling values.
- Maximum rolling values.

Describe how varying the length of the window changes the approximation to the data.

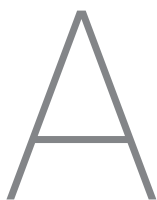


Part II

# Appendices







# Getting Started

The labs in this curriculum aim to introduce computational and mathematical concepts, walk through implementations of those concepts in Python, and use industrial-grade code to solve interesting, relevant problems. Lab assignments are usually about 5–10 pages long and include code examples (yellow boxes), important notes (green boxes), warnings about common errors (red boxes), and about 3–7 exercises (blue boxes). Get started by downloading the lab manual(s) for your course from <http://foundations-of-applied-mathematics.github.io/>.

## Submitting Assignments

### Labs

Every lab has a corresponding specifications file with some code to get you started and to make your submission compatible with automated test drivers. Like the lab manuals, these materials are hosted at <http://foundations-of-applied-mathematics.github.io/>.

Download the .zip file for your course, unzip the folder, and move it somewhere where it won't get lost. This folder has some setup scripts and a collection of folders, one per lab, each of which contains the specifications file(s) for that lab. See **Student-Materials/wiki/Lab-Index** for the complete list of labs, their specifications and data files, and the manual that each lab belongs to.

#### ACHTUNG!

Do **not** move or rename the lab folders or the enclosed specifications files; if you do, the test drivers will not be able to find your assignment. Make sure your folder and file names match **Student-Materials/wiki/Lab-Index**.

To submit a lab, modify the provided specifications file and use the file-sharing program specified by your instructor (discussed in the next section). The instructor will drop feedback files in the lab folder after grading the assignment. For example, the Introduction to Python lab has the specifications file `PythonIntro/python_intro.py`. To complete that assignment, modify `PythonIntro/python_intro.py` and submit it via your instructor's file-sharing system. After grading, the instructor will create a file called `PythonIntro/PythonIntro_feedback.txt` with your score and some feedback.

## Homework

Non-lab coding homework should be placed in the `_Homework/` folder and submitted like a lab assignment. Be careful to name your assignment correctly so the instructor (and test driver) can find it. The instructor may drop specifications files and/or feedback files in this folder as well.

## Setup

### ACHTUNG!

We strongly recommend using a Unix-based operating system (Mac or Linux) for the labs. Unix has a true bash terminal, works well with git and python, and is the preferred platform for computational and data scientists. It is possible to do this curriculum with Windows, but expect some road bumps along the way.

There are two ways to submit code to the instructor: with git (<http://git-scm.com/>), or with a file-syncing service like Google Drive. Your instructor will indicate which system to use.

## Setup With Git

*Git* is a program that manages updates between an online code repository and the copies of the repository, called *clones*, stored locally on computers. If git is not already installed on your computer, download it at <http://git-scm.com/downloads>. If you have never used git, you might want to read a few of the following resources.

- Official git tutorial: <https://git-scm.com/docs/gittutorial>
- Bitbucket git tutorials: <https://www.atlassian.com/git/tutorials>
- GitHub git cheat sheet: [services.github.com/.../github-git-cheat-sheet.pdf](https://services.github.com/.../github-git-cheat-sheet.pdf)
- GitLab git tutorial: <https://docs.gitlab.com/ce/gitlab-basics/start-using-git.html>
- Codecademy git lesson: <https://www.codecademy.com/learn/learn-git>
- Training video series by GitHub: <https://www.youtube.com/playlist?list=PLg7.../>

There are many websites for hosting online git repositories. Your instructor will indicate which web service to use, but we only include instructions here for setup with Bitbucket.

1. *Sign up.* Create a Bitbucket account at <https://bitbucket.org>. If you use an academic email address (ending in `.edu`, etc.), you will get free unlimited public and private repositories.
2. *Make a new repository.* On the Bitbucket page, click the `+` button from the menu on the left and, under **CREATE**, select **Repository**. Provide a name for the repository, mark the repository as **private**, and make sure the repository type is **Git**. For **Include a README?**, select **No** (if you accidentally include a **README**, delete the repository and start over). Under **Advanced settings**, enter a short description for your repository, select **No forks** under forking, and select **Python** as the language. Finally, click the blue **Create repository** button. Take note of the URL of the webpage that is created; it should be something like <https://bitbucket.org/<name>/<repo>>.

3. *Give the instructor access to your repository.* On your newly created Bitbucket repository page (<https://bitbucket.org/<name>/<repo>> or similar), go to **Settings** in the menu to the left and select **User and group access**, the second option from the top. Enter your instructor's Bitbucket username under **Users** and click **Add**. Select the blue **Write** button so your instructor can read from and write feedback to your repository.
4. *Connect your folder to the new repository.* In a shell application (Terminal on Linux or Mac, or Git Bash (<https://gitforwindows.org/>) on Windows), enter the following commands.

```
# Navigate to your folder.
$ cd /path/to/folder # cd means 'change directory'.

# Make sure you are in the right place.
$ pwd                # pwd means 'print working directory'.
/path/to/folder
$ ls *.md             # ls means 'list files'.
README.md             # This means README.md is in the working directory.

# Connect this folder to the online repository.
$ git init
$ git remote add origin https://<name>@bitbucket.org/<name>/<repo>.git

# Record your credentials.
$ git config --local user.name "your name"
$ git config --local user.email "your email"

# Add the contents of this folder to git and update the repository.
$ git add --all
$ git commit -m "initial commit"
$ git push origin master
```

For example, if your Bitbucket username is `greek314`, the repository is called `acmev1`, and the folder is called `Student-Materials/` and is on the desktop, enter the following commands.

```
# Navigate to the folder.
$ cd ~/Desktop/Student-Materials

# Make sure this is the right place.
$ pwd
/Users/Archimedes/Desktop/Student-Materials
$ ls *.md
README.md

# Connect this folder to the online repository.
$ git init
$ git remote add origin https://greek314@bitbucket.org/greek314/acmev1.git

# Record credentials.
$ git config --local user.name "archimedes"
```

```
$ git config --local user.email "greek314@example.com"

# Add the contents of this folder to git and update the repository.
$ git add --all
$ git commit -m "initial commit"
$ git push origin master
```

At this point you should be able to see the files on your repository page from a web browser. If you enter the repository URL incorrectly in the `git remote add origin` step, you can reset it with the following line.

```
$ git remote set-url origin https://<name>@bitbucket.org/<name>/<repo>.git
```

5. *Download data files.* Many labs have accompanying data files. To download these files, navigate to your clone and run the `download_data.sh` bash script, which downloads the files and places them in the correct lab folder for you. You can also find individual data files through `Student-Materials/wiki/Lab-Index`.

```
# Navigate to your folder and run the script.
$ cd /path/to/folder
$ bash download_data.sh
```

6. *Install Python package dependencies.* The labs require several third-party Python packages that don't come bundled with Anaconda. Run the following command to install the necessary packages.

```
# Navigate to your folder and run the script.
$ cd /path/to/folder
$ bash install_dependencies.sh
```

7. (Optional) *Clone your repository.* If you want your repository on another computer after completing steps 1–4, use the following commands.

```
# Navigate to where you want to put the folder.
$ cd ~/Desktop/or/something/

# Clone the folder from the online repository.
$ git clone https://<name>@bitbucket.org/<name>/<repo>.git <foldername>

# Record your credentials in the new folder.
$ cd <foldername>
$ git config --local user.name "your name"
$ git config --local user.email "your email"

# Download data files to the new folder.
$ bash download_data.sh
```

## Setup Without Git

Even if you aren't using git to submit files, you must install it (<http://git-scm.com/downloads>) in order to get the data files for each lab. Share your folder with your instructor according to their directions, and follow steps 5 and 6 of the previous section to download the data files and install package dependencies.

## Using Git

Git manages the history of a file system through *commits*, or checkpoints. Use `git status` to see the files that have been changed since the last commit. These changes are then moved to the *staging area*, a list of files to save during the next commit, with `git add <filename(s)>`. Save the changes in the staging area with `git commit -m "<A brief message describing the changes>"`.

All of these commands are done within a clone of the repository, stored somewhere on a computer. This repository must be manually synchronized with the online repository via two other git commands: `git pull origin master`, to pull updates from the web to the computer; and `git push origin master`, to push updates from the computer to the web.

Command	Explanation
<code>git status</code>	Display the staging area and untracked changes.
<code>git pull origin master</code>	Pull changes from the online repository.
<code>git push origin master</code>	Push changes to the online repository.
<code>git add &lt;filename(s)&gt;</code>	Add a file or files to the staging area.
<code>git add -u</code>	Add all modified, tracked files to the staging area.
<code>git commit -m "&lt;message&gt;"</code>	Save the changes in the staging area with a given message.
<code>git checkout -- &lt;filename&gt;</code>	Revert changes to an unstaged file since the last commit.
<code>git reset HEAD -- &lt;filename&gt;</code>	Remove a file from the staging area.
<code>git diff &lt;filename&gt;</code>	See the changes to an unstaged file since the last commit.
<code>git diff --cached &lt;filename&gt;</code>	See the changes to a staged file since the last commit.
<code>git config --local &lt;option&gt;</code>	Record your credentials ( <code>user.name</code> , <code>user.email</code> , etc.).

Table A.1: Common git commands.

### NOTE

When pulling updates with `git pull origin master`, your terminal may sometimes display the following message.

```
Merge branch 'master' of https://bitbucket.org/<name>/<repo> into master

# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
~
~
```

This means that someone else (the instructor) has pushed a commit that you do not yet have, while you have also made one or more commits locally that they do not have. This screen, displayed in *vim* ([https://en.wikipedia.org/wiki/Vim\\_\(text\\_editor\)](https://en.wikipedia.org/wiki/Vim_(text_editor))), is asking you to enter a message (or use the default message) to create a *merge commit* that will reconcile both changes. To close this screen and create the merge commit, type `:wq` and press `enter`.

## Example Work Session

Short version:

```
$ cd ~/Desktop/Student-Materials/
$ git pull origin master           # Pull updates.

# Make changes to a file.

# Record the changes in git.
$ git add -u                      # Track changes.
$ git commit -m "Made some changes." # Commit changes.
$ git push origin master          # Push updates.
```

Long version:

```
# Navigate to the clone of the repository.
$ cd ~/Desktop/Student-Materials/

# Pull any updates from the online repository (such as TA feedback).
$ git pull origin master
From https://bitbucket.org/username/repo
 * branch          master      -> FETCH_HEAD
Already up-to-date.

# Work on the labs. For example, modify PythonIntro/python_intro.py.

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    PythonIntro/python_intro.py

# Track the changes with git.
$ git add PythonIntro/python_intro.py
$ git status
On branch master
```

```
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   PythonIntro/python_intro.py

# Commit the changes to the repository with an informative message.
$ git commit -m "Made some changes"
[master fed9b34] Made some changes
 1 file changed, 10 insertion(+) 1 deletion(-)

# Push the changes to the online repository.
$ git push origin master
Counting objects: 3, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 327 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://username@bitbucket.org/username/repo.git
 5742a1b..fed9b34  master -> master

# The changes have been saved and the online repository updated.
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```





# B

## Installing and Managing Python

**Lab Objective:** *One of the great advantages of Python is its lack of overhead: it is relatively easy to download, install, start up, and execute. This appendix introduces tools for installing and updating specific packages and gives an overview of possible environments for working efficiently in Python.*

### Installing Python via Anaconda

A *Python distribution* is a single download containing everything needed to install and run Python, together with some common packages. For this curriculum, we **strongly** recommend using the *Anaconda* distribution to install Python. Anaconda includes IPython, a few other tools for developing in Python, and a large selection of packages that are common in applied mathematics, numerical computing, and data science. Anaconda is free and available for Windows, Mac, and Linux.

Follow these steps to install Anaconda.

1. Go to <https://www.anaconda.com/download/>.
2. Download the **Python 3.6** graphical installer specific to your machine.
3. Open the downloaded file and proceed with the default configurations.

For help with installation, see <https://docs.anaconda.com/anaconda/install/>. This page contains links to detailed step-by-step installation instructions for each operating system, as well as information for updating and uninstalling Anaconda.

#### ACHTUNG!

This curriculum uses Python 3.6, **not** Python 2.7. With the wrong version of Python, some example code within the labs may not execute as intended or result in an error.

### Managing Packages

A *Python package manager* is a tool for installing or updating Python packages, which involves downloading the right source code files, placing those files in the correct location on the machine, and linking the files to the Python interpreter. **Never** try to install a Python package without using a package manager (see <https://xkcd.com/349/>).

## Conda

Many packages are not included in the default Anaconda download but can be installed via Anaconda's package manager, `conda`. See <https://docs.anaconda.com/anaconda/packages/pkg-docs> for the complete list of available packages. When you need to update or install a package, **always** try using `conda` first.

Command	Description
<code>conda install &lt;package-name&gt;</code>	Install the specified package.
<code>conda update &lt;package-name&gt;</code>	Update the specified package.
<code>conda update conda</code>	Update <code>conda</code> itself.
<code>conda update anaconda</code>	Update <b>all</b> packages included in Anaconda.
<code>conda --help</code>	Display the documentation for <code>conda</code> .

For example, the following terminal commands attempt to install and update `matplotlib`.

```
$ conda update conda           # Make sure that conda is up to date.
$ conda install matplotlib     # Attempt to install matplotlib.
$ conda update matplotlib      # Attempt to update matplotlib.
```

See <https://conda.io/docs/user-guide/tasks/manage-pkgs.html> for more examples.

### NOTE

The best way to ensure a package has been installed correctly is to try importing it in IPython.

```
# Start IPython from the command line.
$ ipython
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.

# Try to import matplotlib.
In [1]: from matplotlib import pyplot as plt      # Success!
```

### ACHTUNG!

Be careful not to attempt to update a Python package while it is in use. It is safest to update packages while the Python interpreter is not running.

## Pip

The most generic Python package manager is called `pip`. While it has a larger package list, `conda` is the cleaner and safer option. Only use `pip` to manage packages that are not available through `conda`.

Command	Description
<code>pip install package-name</code>	Install the specified package.
<code>pip install --upgrade package-name</code>	Update the specified package.
<code>pip freeze</code>	Display the version number on all installed packages.
<code>pip --help</code>	Display the documentation for pip.

See [https://pip.pypa.io/en/stable/user\\_guide/](https://pip.pypa.io/en/stable/user_guide/) for more complete documentation.

## Workflows

There are several different ways to write and execute programs in Python. Try a variety of workflows to find what works best for you.

### Text Editor + Terminal

The most basic way of developing in Python is to write code in a text editor, then run it using either the Python or IPython interpreter in the terminal.

There are many different text editors available for code development. Many text editors are designed specifically for computer programming which contain features such as syntax highlighting and error detection, and are highly customizable. Try installing and using some of the popular text editors listed below.

- Atom: <https://atom.io/>
- Sublime Text: <https://www.sublimetext.com/>
- Notepad++ (Windows): <https://notepad-plus-plus.org/>
- Geany: <https://www.geany.org/>
- Vim: <https://www.vim.org/>
- Emacs: <https://www.gnu.org/software/emacs/>

Once Python code has been written in a text editor and saved to a file, that file can be executed in the terminal or command line.

```
$ ls                                # List the files in the current directory.
hello_world.py
$ cat hello_world.py                # Print the contents of the file to the terminal.
print("hello, world!")
$ python hello_world.py             # Execute the file.
hello, world!

# Alternatively, start IPython and run the file.
$ ipython
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: %run hello_world.py
hello, world!
```

IPython is an enhanced version of Python that is more user-friendly and interactive. It has many features that cater to productivity such as tab completion and object introspection.

#### NOTE

While Mac and Linux computers come with a built-in bash terminal, Windows computers do not. Windows does come with *Powershell*, a terminal-like application, but some commands in Powershell are different than their bash analogs, and some bash commands are missing from Powershell altogether. There are two good alternatives to the bash terminal for Windows:

- Windows subsystem for linux: [docs.microsoft.com/en-us/windows/wsl/](https://docs.microsoft.com/en-us/windows/wsl/).
- Git bash: <https://gitforwindows.org/>.

## Jupyter Notebook

The Jupyter Notebook (previously known as IPython Notebook) is a browser-based interface for Python that comes included as part of the Anaconda Python Distribution. It has an interface similar to the IPython interpreter, except that input is stored in cells and can be modified and re-evaluated as desired. See <https://github.com/jupyter/jupyter/wiki/> for some examples.

To begin using Jupyter Notebook, run the command `jupyter notebook` in the terminal. This will open your file system in a web browser in the Jupyter framework. To create a Jupyter Notebook, click the **New** drop down menu and choose **Python 3** under the **Notebooks** heading. A new tab will open with a new Jupyter Notebook.

Jupyter Notebooks differ from other forms of Python development in that notebook files contain not only the raw Python code, but also formatting information. As such, Jupyter Notebook files cannot be run in any other development environment. They also have the file extension `.ipynb` rather than the standard Python extension `.py`.

Jupyter Notebooks also support Markdown—a simple text formatting language—and  $\text{\LaTeX}$ , and can embed images, sound clips, videos, and more. This makes Jupyter Notebook the ideal platform for presenting code.

## Integrated Development Environments

An *integrated development environment* (IDEs) is a program that provides a comprehensive environment with the tools necessary for development, all combined into a single application. Most IDEs have many tightly integrated tools that are easily accessible, but come with more overhead than a plain text editor. Consider trying out each of the following IDEs.

- JupyterLab: <http://jupyterlab.readthedocs.io/en/stable/>
- PyCharm: <https://www.jetbrains.com/pycharm/>
- Spyder: <http://code.google.com/p/spyderlib/>
- Eclipse with PyDev: <http://www.eclipse.org/>, <https://www.pydev.org/>

See <https://realpython.com/python-ides-code-editors-guide/> for a good overview of these (and other) workflow tools.



# NumPy Visual Guide

**Lab Objective:** *NumPy operations can be difficult to visualize, but the concepts are straightforward. This appendix provides visual demonstrations of how NumPy arrays are used with slicing syntax, stacking, broadcasting, and axis-specific operations. Though these visualizations are for 1- or 2-dimensional arrays, the concepts can be extended to n-dimensional arrays.*

## Data Access

The entries of a 2-D array are the rows of the matrix (as 1-D arrays). To access a single entry, enter the row index, a comma, and the column index. Remember that indexing begins with 0.

$$A[0] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \quad A[2,1] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

## Slicing

A lone colon extracts an entire row or column from a 2-D array. The syntax `[a:b]` can be read as “the *a*th entry up to (but not including) the *b*th entry.” Similarly, `[a:]` means “the *a*th entry to the end” and `[:b]` means “everything up to (but not including) the *b*th entry.”

$$A[1] = A[1,:] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \quad A[:,2] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$
$$A[1:,:2] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \quad A[1:-1,1:-1] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

## Stacking

`np.hstack()` stacks sequence of arrays horizontally and `np.vstack()` stacks a sequence of arrays vertically.

$$\begin{aligned}
 A &= \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix} & B &= \begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \end{bmatrix} \\
 \\ 
 \text{np.hstack}((A,B,A)) &= \begin{bmatrix} \times & \times & \times & * & * & * & \times & \times & \times \\ \times & \times & \times & * & * & * & \times & \times & \times \\ \times & \times & \times & * & * & * & \times & \times & \times \end{bmatrix} \\
 \\ 
 \text{np.vstack}((A,B,A)) &= \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ * & * & * \\ * & * & * \\ * & * & * \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix}
 \end{aligned}$$

Because 1-D arrays are flat, `np.hstack()` concatenates 1-D arrays and `np.vstack()` stacks them vertically. To make several 1-D arrays into the columns of a 2-D array, use `np.column_stack()`.

$$\begin{aligned}
 x &= [\times \quad \times \quad \times \quad \times] & y &= [* \quad * \quad * \quad *] \\
 \\ 
 \text{np.hstack}((x,y,x)) &= [\times \quad \times \quad \times \quad \times \quad * \quad * \quad * \quad * \quad \times \quad \times \quad \times \quad \times] \\
 \\ 
 \text{np.vstack}((x,y,x)) &= \begin{bmatrix} \times & \times & \times & \times \\ * & * & * & * \\ \times & \times & \times & \times \end{bmatrix} & \text{np.column_stack}((x,y,x)) &= \begin{bmatrix} \times & * & \times \\ \times & * & \times \\ \times & * & \times \\ \times & * & \times \end{bmatrix}
 \end{aligned}$$

The functions `np.concatenate()` and `np.stack()` are more general versions of `np.hstack()` and `np.vstack()`, and `np.row_stack()` is an alias for `np.vstack()`.

## Broadcasting

NumPy automatically aligns arrays for component-wise operations whenever possible. See <http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html> for more in-depth examples and broadcasting rules.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} \qquad x = [ 10 \quad 20 \quad 30 ]$$

$$A + x = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 10 & 20 & 30 \end{bmatrix} = \begin{bmatrix} 11 & 22 & 33 \\ 11 & 22 & 33 \\ 11 & 22 & 33 \end{bmatrix}$$

$$A + x.\text{reshape}((1,-1)) = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix} = \begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{bmatrix}$$

## Operations along an Axis

Most array methods have an **axis** argument that allows an operation to be done along a given axis. To compute the sum of each column, use **axis=0**; to compute the sum of each row, use **axis=1**.

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

$$A.\text{sum}(\text{axis}=0) = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} = [ 4 \quad 8 \quad 12 \quad 16 ]$$

$$A.\text{sum}(\text{axis}=1) = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} = [ 10 \quad 10 \quad 10 \quad 10 ]$$