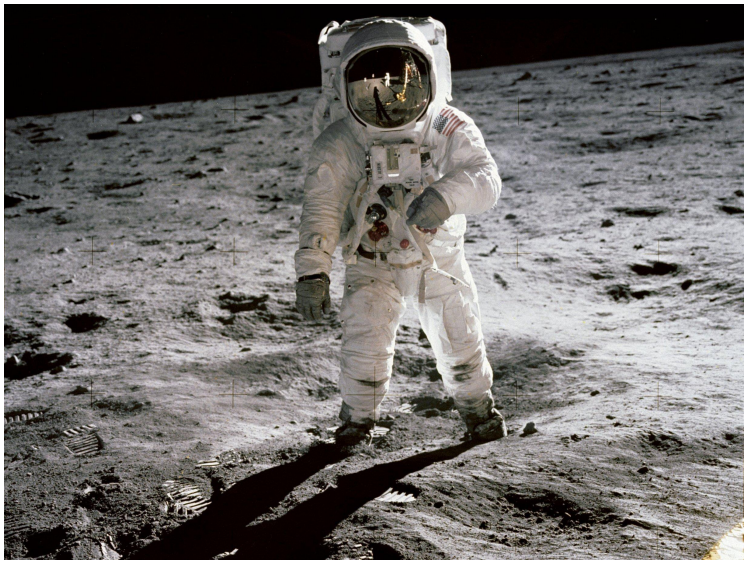
Labs for Foundations of Applied Mathematics

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

C. Carter
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

K. Finlinson
Brigham Young University

J. Fisher
Brigham Young University

R. Flores
Brigham Young University

R. Fowers
Brigham Young University

A. Frandsen
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

E. Hannesson
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

H. Ringer
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

This project is funded in part by the National Science Foundation, grant no. TUES Phase II DUE-1323785.

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. The reader should be familiar with Python [VD10] and its NumPy [Oli06, ADH+01, Oli07] and Matplotlib [Hun07] packages before attempting these labs. See the Python Essentials manual for introductions to these topics.

Contents

Preface iii

I Labs 1

1 Introduction to the Unix Shell 3

2 Unix Shell 2 17

3 SQL 1: Introduction 31

4 SQL 2 (The Sequel) 43

5 Regular Expressions 51

6 Web Technologies 63

7 Web Scraping 75

8 Web Crawling 85

9 Pandas 1: Introduction 93

10 Pandas 2: Plotting 107

11 Pandas 3: Grouping 121

12 Pandas 4: Time Series 133

13 Geopandas 147

14 MongoDB 155

15 Intro to Parallel Computing 165

16 Parallel Programming with MPI 177

17 Introduction to Apache Spark 185

II Appendices **193**

A Getting Started **195**

B Installing and Managing Python **203**

C NumPy Visual Guide **207**

D Introduction to Scikit-Learn **211**

Bibliography **227**

Part I

Labs

1 Unix Shell 1: Introduction

Lab Objective: *Unix is a popular operating system that is commonly used for servers and the basis for most open source software. Using Unix for writing and submitting labs will develop a foundation for future software development. In this lab we explore the basics of the Unix shell, including how to navigate and manipulate files, access remote machines with Secure Shell, and use Git for basic version control.*

Unix was first developed by AT&T Bell Labs in the 1970s. In the 1990s, Unix became the foundation of the Linux and MacOSX operating systems. Most servers are Linux-based, so knowing how to use Unix shells allows us to interact with servers and other Unix-based machines.

A *Unix shell* is a program that takes commands from a user and executes those commands on the operating system. We interact with the shell through a *terminal* (also called a command line), a program that lets you type in commands and gives those commands to the shell for execution.

> **Note**
>
> Windows is not built off of Unix, but it does come with a terminal called PowerShell. This terminal uses a different command syntax. We will not cover the equivalent commands in the Windows terminal, but you could download a Unix-based terminal such as Git Bash or Cygwin to complete this lab on a Windows machine (you will still lose out on certain commands). Alternatively, Windows 10 now offers a Windows Subsystem for Linux, WSL, which is a Linux operating system downloaded onto Windows.

> **Note**
>
> For this lab we will be working in the `UnixShell1` directory provided with the lab materials. If you have not yet downloaded the code repository, follow steps 1 through 6 in the `Getting Started` guide found at `https://foundations-of-applied-mathematics.github.io/` before proceeding with this lab. Make sure to run the `download_data.sh` script as described in step 5 of `Getting Started`; otherwise you will not have the necessary files to complete this lab.

Basic Unix Shell

Shell Scripting

The following sections of the lab will explore several shell commands. You can execute these commands by typing these commands directly into a terminal. Sometimes, though, you will want to execute a more complicated sequence of commands, or make it easy to execute the same set of commands over and over again. In those cases, it is useful to create a *script*, which is a sequence of shell commands saved in a file. Then, instead of typing the commands in individually, you simply have to run the script, and it takes care of running all the commands.

In this lab we will be running and editing a bash script. Bash is the most commonly used Unix shell and is the default shell installed on most Unix-based systems.

The following is a very simple bash script. The command `echo <string>` prints `<string>` in the terminal.

```
#!/bin/bash
echo "Hello World!"
```

The first line, `#!bin/bash`, tells the computer to use the bash interpreter to run the script, and where this interpreter is located. The `#!` is called the *shebang* or *hashbang* character sequence. It is followed by the absolute path to the bash interpreter.

To run a bash script, type `bash <script name>` into the terminal. Alternatively, you can execute any script by typing `./<script name>`, but note that the script must contain executable permissions for this to work. (We will learn more about permissions later in the lab.)

```
$ bash hello_world.sh
Hello World!
```

Navigation

Typically, people navigate computers by clicking on icons to open folders and programs. In the terminal, instead of point and click we use typed commands to move from folder to folder. In the Unix shell, we call folders *directories*. The file system is a set of nested directories containing files and other directories.

Begin by opening a terminal. The text you see in the upper left of the terminal is called the *prompt*. Before you start creating or deleting files, you'll want to know where you are. To see what directory you are currently working in, type `pwd` into the prompt. This command stands for **p**rint **w**orking **d**irectory, and it prints out a string telling you your current location.

To see the all the contents of your current directory, type the command `ls`, **l**ist **s**egments.

```
~$ pwd
/home/username

~$ ls
Desktop     Downloads       Public      Videos
Documents   Pictures
```

The command `cd`, **c**hange **d**irectory, allows you to navigate directories. To change to a new directory, type the `cd` command followed by the name of the directory to which you want to move (if you `cd` into a file, you will get an error). You can move up one directory by typing `cd ..`.

Two important directories are the root directory and the home directory. You can navigate to the home directory by typing `cd ~` or just `cd`. You can navigate to root by typing `cd /`.

Problem 1. To begin, open a terminal and navigate to the `UnixShell1/` directory provided with this lab. Use `ls` to list the contents. There should be a file called `Shell1.zip` and a script called `unixshell1.sh`. [a]

Run `unixshell1.sh`. This script will do the following:

1. Unzip `Shell1.zip`, creating a directory called `Shell1/`
2. Remove any previously unzipped copies of `Shell1/`
3. Execute various shell commands, to be added in the next few problems in this lab
4. Create a compressed version of `Shell1/` called `UnixShell1.tar.gz`.
5. Remove any old copies of `UnixShell1.tar.gz`

Now, open the `unixshell1.sh` script in a text editor. Add commands to the script to do the following:

- Change into the `Shell1/` directory.
- Print a string telling you directory you are currently working in.

Test your commands by running the script again and checking that it prints a string ending in the location `Shell1/`.

[a] If the necessary data files are not in your directory, `cd` one directory up by typing `cd ..` and type `bash download_data.sh` to download the data files for each lab.

Documentation and Help

When you encounter an unfamiliar command, the terminal has several tools that can help you understand what it does and how to use it. Most commands have manual pages, which give information about what the command does, the syntax required to use it, and different options to modify the command. To open the manual page for a command, type `man <command>`. Some commands also have an option called `--help`, which will print out information similar to what is contained in the manual page. To use this option, type `<command> --help`.

```
$ man ls
LS(1)                          User Commands                          LS(1)

NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...
```

```
DESCRIPTION
       List  information  about  the FILEs (the current directory by default).

       -a, --all
              do not ignore entries starting with .
```

The `apropos <keyword>` command will list all Unix commands that have `<keyword>` contained somewhere in their manual page names and descriptions. For example, if you forget how to copy files, you can type in `apropos copy` and you'll get a list of all commands that have `copy` in their description.

Flags

When you use `man`, you will see a list of options such as `-a`, `-A`, `--author`, etc. that modify how a command functions. These are called *flags*. You can use one flag on a command by typing `<command> -<flag>`, like `ls -a`, or combine multiple flags by typing `<command> -<flag1><flag2>`, etc. as in `ls -alt`.

For example, sometimes directories contain hidden files, which are files whose names begin with a dot character like `.bash`. The `ls` command, by default, does not list hidden files. Using the `-a` flag specifies that `ls` should not ignore hidden files. Find more common flags for `ls` in Table 1.1.

Flags	Description
`-a`	Do not ignore hidden files and folders
`-l`	List files and folders in long format
`-r`	Reverse order while sorting
`-R`	Print files and subdirectories recursively
`-s`	Print item name and size
`-S`	Sort by size
`-t`	Sort output by date modified

Table 1.1: Common flags of the `ls` command.

```
$ ls
file1.py  file2.py

$ ls -a
.  ..  file1.py  file2.py  .hiddenfile.py

$ ls -alt    # Multiple flags can be combined into one flag
total 8
drwxr-xr-x  2 c c 4096 Aug 14 10:08 .
-rw-r--r--  1 c c    0 Aug 14 10:08 .hiddenfile.py
-rw-r--r--  1 c c    0 Aug 14 10:08 file2.py
-rw-r--r--  1 c c    0 Aug 14 10:08 file1.py
drwxr-xr-x 38 c c 4096 Aug 14 10:08 ..
```

Problem 2. Within the script, add a command using `ls` to print one list of the contents of `Shell1/` with the following criteria:

- Include hidden files and folders
- List the files and folders in long format (include the permissions, date last modified, etc.)
- Sort the output by file size (largest files first)

Test your command by entering it into the terminal within `Shell1/` or by running the script and checking for the desired output.

Manipulating Files and Directories

In this section we will learn how to create, copy, move, and delete files and folders. To create a text file, use `touch <filename>`. To create a new directory, use `mkdir <dir_name>`.

```
~$ cd Test/                 # navigate to test directory

~/Test$ ls                  # list contents of directory
file1.py

~/Test$ mkdir NewDirectory  # create a new empty directory

~/Test$ touch newfile.py    # create a new empty file

~/Test$ ls
file1.py  NewDirectory  newfile.py
```

To copy a file into a directory, use `cp <filename> <dir_name>`. When making a copy of a directory, use the `-r` flag to recursively copy files contained in the directory. If you try to copy a directory without the `-r`, the command will return an error.

Moving files and directories follows a similar format, except no `-r` flag is used when moving one directory into another. The command `mv <filename> <dir_name>` will move a file to a folder and `mv <dir1> <dir2>` will move the first directory into the second.

If you want to rename a file, use `mv <file_old> <file_new>`; the same goes for directories.

```
~/Test$ ls
file1.py  NewDirectory  newfile.py

~/Test$ mv newfile.py NewDirectory/         # move file into directory
~/Test$ cp file1.py NewDirectory/           # make a copy of file1 in directory
~/Test$ cd NewDirectory/
~/Test/NewDirectory$ mv file1.py newname.py # rename file1.py
~/Test/NewDirectory$ ls
newfile.py  newname.py
```

When deleting files, use `rm <filename>`, and when deleting a directory, use `rm -r <dir_name>`. The `-r` flag tells the terminal to recursively remove all the files and subfolders within the targeted directory.

If you want to make sure your command is doing what you intend, the `-v` flag tells `rm`, `cp`, or `mkdir` to print strings in the terminal describing what it is doing.

When your terminal gets too cluttered, use `clear` to clean it up.

```
~/Test/NewDirectory$ cd ..          # move one directory up
~/Test$ rm -rv NewDirectory/        # remove a directory and its contents
removed 'NewDirectory/newname.py'
removed 'NewDirectory/newfile.py'
removed directory 'NewDirectory/'

~/Test$ rm file1.py                 # remove a file
~/Test$ ls                          # directory is now empty
~/Test$
```

Commands	Description
`clear`	Clear the terminal screen
`cp file1 dir1`	Create a copy of `file1` and move it to `dir1/`
`cp file1 file2`	Create a copy of `file1` and name it `file2`
`cp -r dir1 dir2`	Create a copy of `dir1/` and all its contents into `dir2/`
`mkdir dir1`	Create a new directory named `dir1/`
`mkdir -p path/to/new/dir1`	Create `dir1/` and all intermediate directories
`mv file1 dir1`	Move `file1` to `dir1/`
`mv file1 file2`	Rename `file1` as `file2`
`rm file1`	Delete `file1` [`-i`, `-v`]
`rm -r dir1`	Delete `dir1/` and all items within `dir1/` [`-i`, `-v`]
`touch file1`	Create an empty file named `file1`

Table 1.2: File Manipulation Commands

Table 1.2 contains all the commands we have discussed so far. Commonly used flags for some commands are contained in square brackets; use `man` or `--help` to see what these mean.

Problem 3. Add commands to the `unixshell1.sh` script to make the following changes in `Shell1/`:

- Delete the `Audio/` directory along with all its contents
- Create `Documents/`, `Photos/`, and `Python/` directories
- Change the name of the `Random/` directory to `Files/`

Test your commands by running the script and then using `ls` within `Shell1/` to see what directories are there. Once you have run the script and deleted, created or renamed files,

Wildcards

As we are working in the file system, there will be times that we want to perform the same command to a group of similar files. For example, you may need to move all text files within a directory to a new directory. Rather than copy each file one at a time, we can apply one command to several files using *wildcards*. We will use the `*` and `?` wildcards. The `*` wildcard represents any string and the `?` wildcard represents any single character. Though these wildcards can be used in almost every Unix command, they are particularly useful when dealing with files.

```
$ ls
File1.txt   File2.txt   File3.jpg   text_files

$ mv -v *.txt text_files/
File1.txt -> text_files/File1.txt
File2.txt -> text_files/File2.txt

$ ls
File3.jpg   text_files
```

See Table 1.3 for examples of common wildcard usage.

Command	Description
`*.txt`	All files that end with `.txt`.
`image*`	All files that have `image` as the first 5 characters.
`*py*`	All files that contain `py` in the name.
`doc*.txt`	All files of the form `doc1.txt`, `doc2.txt`, `docA.txt`, etc.

Table 1.3: Common uses for wildcards.

Problem 4. Within the `Shell1/` directory, there are many files. Add commands to the script to organize these files into directories using wildcards. Organize by completing the following:

- Move all the `.jpg` files to the `Photos/` directory
- Move all the `.txt` files to the `Documents/` directory
- Move all the `.py` files to the `Python/` directory

Working With Files

Searching the File System

There are two commands we can use for searching through our directories. The `find` command is used to find files or directories with a certain name; the `grep` command is used to find lines within files matching a certain string. When searching for a specific string, both commands allow wildcards within the string. You can use wildcards so that your search string matches a broader set of strings.

```
# Find all files or directories in Shell1/ called "final"
# -type f,d specifies to look for files and directories
# . specifies to look in the current directory

$ find . -name "final" -type f,d
$           # There are no files with the exact name "final" in Shell1/

$ find . -name "*final*" -type f,d
./Files/May/finals
./Files/May/finals/finalproject.py
```

```
# Find all within files in Documents/ containing "Mary"
# -r tells grep to search all files with Documents/
# -n tells grep to print out the line number (2)

$ Shell1$ grep -nr "Mary" Documents/
Documents/people.txt:2:female,Mary,31
```

Command	Description
`find dir1 -type f -name "word"`	Find all files in `dir1/` (and its subdirectories) called `word` (`-type f` is for files; `-type d` is for directories)
`grep "word" filename`	Find all occurences of `word` within `filename`
`grep -nr "word" dir1`	Find all occurences of `word` within the files inside `dir1/` (`-n` lists the line number; `-r` performs a recursive search)

Table 1.4: Commands using `find` and `grep`.

Table 1.4 contains basic sytax for using these two commands. There are many more variations of syntax for `grep` and `find`, however. You can use `man grep` and `man find` to explore other options for using these commands.

File Security and Permissions

A file has three levels of permissions associated with it: the permission to read the file, to write (modify) the file, and to execute the file. There are also three categories of people who are assigned permissions: the user (the owner), the group, and others.

You can check the permissions for `file1` using the command `ls -l <file1>`. Note that your output will differ from that printed below; this is purely an example.

```
$ ls -l
-rw-rw-r-- 1 username groupname 194 Aug  5 20:20 calc.py
drw-rw-r-- 1 username groupname 373 Aug  5 21:16 Documents
-rwxr-x--x 1 username groupname  27 Aug  5 20:22 mult.py
-rw-rw-r-- 1 username groupname 721 Aug  5 20:23 project.py
```

The first character of each line denotes the type of the item whether it be a normal file, a directory, a symbolic link, etc. The next nine characters denote the permissions associated with that file.

For example, look at the output for `mult.py`. The first character `-` denotes that `mult.py` is a normal file. The next three characters, `rwx`, tell us the owner can read, write, and execute the file. The next three characters, `r-x`, tell us members of the same group can read and execute the file, but not edit it. The final three characters, `--x`, tell us other users can execute the file and nothing more.

Permissions can be modified using the `chmod` command. There are multiple notations used to modify permissions, but the easiest to use when we want to make small modifications to a file's permissions is *symbolic permissions* notation. See Table 1.5 for more examples of using symbolic permissions notation, as well as other useful commands for working with permissions.

```
$ ls -l script1.sh
total 0
-rw-r--r-- 1 c c 0 Aug 21 13:06 script1.sh

$ chmod u+x script1.sh      # add permission for user to execute
$ chmod o-r script1.sh      # remove permission for others to read
$ ls -l script1.sh
total 0
-rwxr----- 1 c c 0 Aug 21 13:06 script1.sh
```

Command	Description
`chmod u+x file1`	Add executing (`x`) permissions to user (`u`)
`chmod g-w file1`	Remove writing (`w`) permissions from group (`g`)
`chmod o-r file1`	Remove reading (`r`) permissions from other other users (`o`)
`chmod a+w file1`	Add writing permissions to everyone (`a`)
`chown`	change owner
`chgrp`	change group
`getfacl`	view all permissions of a file in a readable format.

Table 1.5: Symbolic permissions notation and other useful commands

Running Files

To run a file for which you have execution permissions, type the file name preceded by `./`.

```
$ ./hello.sh
bash: ./hello.sh: Permission denied

$ ls -l hello.sh
-rw-r--r-- 1 username groupname 31 Jul 30 14:34 hello.sh

$ chmod u+x hello.sh        # You can now execute the file

$ ./hello.sh
Hello World!
```

Problem 5. Within `Shell1/`, there is a script called `organize_photos.sh`. First, use `find` to locate the script. Once you know the file location, add commands to your script so that it completes the following tasks:

- Moves `organize_photos.sh` to `Scripts/`
- Adds executable permissions to the script for the user
- Runs the script

Test that the script has been executed by checking that additional files have been moved into the `Photos/` directory. Check that permissions have been updated on the script by using `ls -l`.

Accessing Remote Machines

At times you will find it useful to perform tasks on a remote computer or server, such as running a script that requires a large amount of computing power on a supercomputer or accessing a data file stored on another machine.

Secure Shell

Secure Shell (SSH) allows you to remotely access other computers or servers securely. SSH is a network protocol encrypted using public-key cryptography. It ensures that all communication between your computer and the remote server is secure and encrypted.

The system you are connecting to is called the *host*, and the system you are connecting from is called the *client*. The first time you connect to a host, you will receive a warning saying the authenticity of the host can't be established. This warning is a default, and appears when you are connecting to a host you have not connected to before. When asked if you would like to continue connecting, select yes.

When prompted for your password, type your password as normal and press enter. No characters will appear on the screen, but they are still being logged. Once the connection is established, there is a secure tunnel through which commands and files can be exchanged between the client and host. To end a secure connection, type exit.

```
alice@mycomputer:~$ ssh alice27@acme01.byu.edu

alice27@acme01.byu.edu password:# Type password as normal
last login 7 Sept 11

[alice27@byu.local@acme01 ~]$ ls    # Commands are executed on the host
myacmeshare/

[alice27@byu.local@acme01 ~]$ exit  # End a secure connection
logout
Connection to acme01.byu.edu closed.

alice@mycomputer:~$                 # Commands are executed on the client
```

Secure Copy

To copy files from one computer to another, you can use the Unix command `scp`, which stands for **s**ecure **c**opy **p**rotocol. The syntax for `scp` is essentially the same as the syntax for `cp`.

To copy a file from your computer to a specific location on on a remote machine, use the syntax `scp <file1> <user@remote_host:file_path>`. As with `cp`, to copy a directory and all of its contents, use the `-r` flag.

```
# Make copies of file1 and dir2 in the home directory on acme01.byu.edu
alice@mycomputer:~$ scp file1 alice27@acme01.byu.edu:~/
alice@mycomputer:~$ scp -r dir1/dir2 alice27@acme01.byu.edu:~/
```

Use the syntax `scp -r <user@remote_host:file_path/dir1> <file_path>` to copy `dir1` from a remote machine to the location specified by `file_path` on your current machine.

```
# Make a local copy of dir1 (from acme01.byu.edu) in the home directory
alice@mycomputer:~$ scp -r alice27@acme01.byu.edu:~/dir1  ~
```

Commands	Description
`ssh username@remote_host`	Establish a secure connection with `remote_host`
`scp file1 user@remote_host:file_path/`	Create a copy of `file1` on host
`scp -r dir1 user@remote_host:file_path/`	Create a copy of `dir1` and its contents on host
`scp user@remote_host:file_path/file1 file_path2`	Create a local copy of file on client

Table 1.6: Basic syntax for `ssh` and `scp`.

Problem 6. On a computer with the host name `acme20.byu.edu` or `acme21.byu.edu`, there is a file called `img_649.jpg`. Secure copy this file to your `UnixShell1/` directory. (Do not add the `scp` command to the script).

To `ssh` or `scp` on this computer, your username is your Net ID, and your password is your typical Net ID password. To use `scp` or `ssh` for this computer, you will have to be on campus using BYU Wifi.

Hint: To use `scp`, you will need to know the location of the file on the remote computer. Consider using `ssh` to access the machine and using `find`. The file is located somewhere in the directory `/sshlab`.

After secure copying, add a command to your script to copy the file from `UnixShell1/` into the directory `Shell1/Photos/`. (Make sure to leave a copy of the file in `UnixShell1/`, otherwise the file will be deleted when you run the script again.)

Git

Git is a version control system, meaning that it keeps a record of changes in a file. Git also facilitates collaboration between people working on the same code. It does both these things by managing updates between an online code repository and copies of the repository, called *clones*, stored locally on computers.

We will be using git to submit labs and return feedback on those labs. If git is not already installed on your computer, download it at `http://git-scm.com/downloads`.

Using Git

Git manages the history of a file system through *commits*, or checkpoints. Each time a new commit is added to the online repository, a checkpoint is created so that if need be, you can use or look back at an older version of the repository. You can use `git log` to see a list of previous commits. You can also use `git status` to see the files that have been changed in your local repository since the last commit.

Before making your own changes, you'll want to add any commits from other clones into your local repository. To do this, use the command `git pull origin master`.

Once you have made changes and want to make a new commit, there are normally three steps. To save these changes to the online repository, first add the changed files to the *staging area*, a list of files to save during the next commit, with `git add <filename(s)>`. If you want to add all changes that you have made to tracked files (files that are already included in the online repository), use `git add -u`.

Next, save the changes in the staging area with `git commit -m "<A brief message describing the changes>"`.

Finally, add the changes in this commit to the online repository with `git push origin master`.

```
$ cd MyDirectory/               # Navigate into a cloned repository
$ git pull origin master        # Pull new commits from online repository

### Make changes to file1.py ###

$ git add file1.py              # Add file to staging area
$ git commit -m "Made changes"  # Commit changes in staging area
$ git push origin master        # Push changes to online repository
```

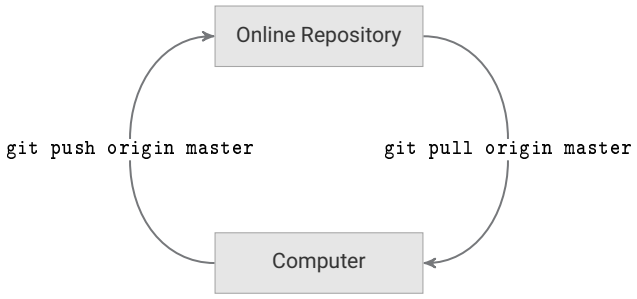


Figure 1.1: Exchanging git commits between the repository and a local clone.

Merge Conflicts

Git maintains order by raising an alert when changes are made to the same file in different clones and neither clone contains the changes made in the other. This is called a *merge conflict*, which happens when someone else 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.

Achtung!

When pulling updates with `git pull origin master`, your terminal may sometimes display the following merge conflict 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 screen, displayed in `vim` (`https://en.wikipedia.org/wiki/Vim_(text_editor)`), is asking you to enter a message to create a *merge commit* that will reconcile both changes. If you do not enter a message, a default message is used. To close this screen and create the merge commit with the default message, type `:wq` (the characters will appear in the bottom left corner of the terminal) and press `enter`.

Note

`Vim` is a terminal text editor available on essentially any computer you will use. When working with remote machines through `ssh`, vim is often the only text editor available to use. To exit vim, press `esc:wq` To learn more about vim, visit the official documentation at `https://vimhelp.org`.

Command	Explanation
`git status`	Display the staging area and untracked changes.
`git pull origin master`	Pull changes from the online repository.
`git push origin master`	Push changes to the online repository.
`git add <filename(s)>`	Add a file or files to the staging area.
`git add -u`	Add all modified, tracked files to the staging area.
`git commit -m "<message>"`	Save the changes in the staging area with a given message.
`git checkout <filename>`	Revert changes to an unstaged file since the last commit.
`git reset HEAD <filename>`	Remove a file from the staging area, but keep changes.
`git diff <filename>`	See the changes to an unstaged file since the last commit.
`git diff --cached <filename>`	See the changes to a staged file since the last commit.
`git config --local <option>`	Record your credentials (`user.name`, `user.email`, etc.).

Table 1.7: Common git commands.

Problem 7. Using git commands, push `unixshell1.sh` and `UnixShell1.tar.gz` to your online git repository. Do not add anything else in the `UnixShell1/` directory to the online repository.

2 Unix Shell 2

Lab Objective: *Introduce system management, calling Unix Shell commands within Python, and other advanced topics. As in the last lab, the majority of learning will not be had in finishing the problems, but in following the examples.*

File Security

To begin, run the following command while inside the `Shell2/Python/` directory (`Shell2/` is the end product of `Shell1/` from the previous lab). Notice your output will differ from that printed below; this is for learning purposes.

```
$ ls -l
-rw-rw-r-- 1 username groupname 194 Aug  5 20:20 calc.py
-rw-rw-r-- 1 username groupname 373 Aug  5 21:16 count_files.py
-rwxr-xr-x 1 username groupname  27 Aug  5 20:22 mult.py
-rw-rw-r-- 1 username groupname 721 Aug  5 20:23 project.py
```

Notice the first column of the output. The first character denotes the type of the item whether it be a normal file, a directory, a symbolic link, etc. The remaining nine characters denote the permissions associated with that file. Specifically, these permissions deal with reading, wrtiting, and executing files. There are three categories of people associated with permissions. These are the user (the owner), group, and others. For example, look at the output for `mult.py`. The first character `-` denotes that `mult.py` is a normal file. The next three characters, `rwx` tell us the owner can read, write, and execute the file. The next three characters `r-x` tell us members of the same group can read and execute the file. The final three characters `--x` tell us other users can execute the file and nothing more.

Permissions can be modified using the `chmod` command. There are two different ways to specify permissions, *symbolic permissions* notation and *octal permissions* notation. Symbolic permissions notation is easier to use when we want to make small modifications to a file's permissions. See Table 2.1.

Octal permissions notation is easier to use when we want to set all the permissions as once. The number 4 corresponds to reading, 2 corresponds to writing, and 1 corresponds to executing. See Table 2.2.

The commands in Table 2.3 are also helpful when working with permissions.

Command	Description
`chmod u+x file1`	Add executing (`x`) permissions to user (`u`)
`chmod g-w file1`	Remove writing (`w`) permissions from group (`g`)
`chmod o-r file1`	Remove reading (`r`) permissions from other other users (`o`)
`chmod a+w file1`	Add writing permissions to everyone (`a`)

Table 2.1: Symbolic permissions notation

Command	Description
`chmod 760 file1`	Sets `rwx` to user, `rw-` to group, and `---` to others
`chmod 640 file1`	Sets `rw-` to user, `r--` to group, and `---` to others
`chmod 775 file1`	Sets `rwx` to user, `rwx` to group, and `r-x` to others
`chmod 500 file1`	Sets `r-x` to user, `---` to group, and `---` to others

Table 2.2: Octal permissions notation

Scripts

A shell script is a series of shell commands saved in a file. Scripts are useful when we have a process that we do over and over again. The following is a very simple script.

```
#!/bin/bash
echo "Hello World"
```

The first line starts with `"#!"`. This is called the *shebang* or *hashbang* character sequence. It is followed by the absolute path to the `bash` interpreter. If we were unsure where the `bash` interpreter is saved, we run `which bash`.

Problem 1. Create a file called `hello.sh` that contains the previous text and save it in the `Scripts/` directory. The file extension `.sh` is technically unnecessary, but it is good practice to always include an extension.

To execute a script, type the script name preceded by `./`

```
$ cd Scripts
$ ./hello.sh
bash: ./hello.sh: Permission denied

# Notice you do not have permission to execute this file. This is by default.
$ ls -l hello.sh
-rw-r--r-- 1 username groupname 31 Jul 30 14:34 hello.sh
```

Problem 2. Add executable permissions to `hello.sh`. Run the script to verify that it worked.

You can do this same thing with Python scripts. All you have to do is change the path following the shebang. To see where the Python interpreter is stored, run `which python`.

Command	Description
`chown`	change owner
`chgrp`	change group
`getfacl`	view all permissions of a file in a readable format.

Table 2.3: Other commands when working with permissions

Command	Description
`df dir1`	Display available disk space in file system containing `dir1`
`du dir1`	Display disk usage within `dir1` [`-a`, `-h`]
`free`	Display amount of free and used memory in the system
`ps`	Display a snapshot of current processes
`top`	Display interactive list of current processes

Table 2.4: Commands for resource management

Problem 3. The `Python/` directory contains a file called `count_files.py`, a Python script that counts all the files within the `Shell2/` directory. Modify this file so it can be run as a script and change the permissions of this script so the user and group can execute the script.

NOTE

If you would like to learn how to run scripts on a set schedule, consider researching *cron jobs*.

Resource Management

To be able to optimize performance, it is valuable to always be aware of the resources we are using. Hard drive space and computer memory are two resources we must constantly keep in mind. The commands found in Table 2.4 are essential to managing resources.

Job Control

Let's say we had a series of scripts we wanted to run. If we knew that these would take a while to execute, we may want to start them all at the same time and let them run while we are working on something else. In Table 2.5, we have listed some of the most common commands used in job control. We strongly encourage you to experiment with these commands.

Command	Description
`COMMAND &`	Adding an ampersand to the end of a command runs the command in the background
`bg %N`	Restarts the Nth interrupted job in the background
`fg %N`	Brings the Nth job into the foreground
`jobs`	Lists all the jobs currently running
`kill %N`	Terminates the Nth job
`ps`	Lists all the current processes
`Ctrl-C`	Terminates current job
`Ctrl-Z`	Interrupts current job
`nohup`	Run a command that will not be killed if the user logs out

Table 2.5: Job control commands

The `five_secs.sh` and `ten_secs.sh` scripts in the `Scripts/` directory take five seconds and ten seconds to execute respectively. These will be particularly useful as you are experimenting with these commands.

```
# Don't forget to change permissions if needed
$ ./ten_secs.sh &
$ ./five_secs.sh &
$ jobs
[1]+  Running       ./ten_secs.sh &
[2]-  Running       ./five_secs.sh &
$ kill %2
[2]-  Terminated    ./five_secs.sh &
$ jobs
[1]+  Running       ./ten_secs.sh &
```

Problem 4. In addition to the `five_secs.sh` and `ten_secs.sh` scripts, the `Scripts/` folder contains three scripts that each take about forty-five seconds to execute. Execute each of these commands in the background so all three are running at the same time. While they are all running, write the output of `jobs` to a new file `log.txt` saved in the `Scripts/` directory.

Using Python for File Management

Bash itself has control flow tools like if-else blocks and loops, but most of the syntax is highly unintuitive. Python, on the other hand, has extremely intuitive syntax for these control flow tools, so using Python to do shell-like tasks can result in some powerful but specific file management programs. Table 2.6 relates some of the common shell commands to Python functions, most of which come from the `os` module in the standard library.

In addition to these, Python has a few extra functions that are useful for file management and shell commands. See Table 2.7. The two functions `os.walk()` and `glob.glob()` are especially useful for doing searches like `find` and `grep`.

Shell Command	Python Function
`ls`	`os.listdir()`
`cd`	`os.chdir()`
`pwd`	`os.getcwd()`
`mkdir`	`os.mkdir()`, `os.mkdirs()`
`cp`	`shutil.copy()`
`mv`	`os.rename()`, `os.replace()`
`rm`	`os.remove()`, `shutil.rmtree()`
`du`	`os.path.getsize()`
`chmod`	`os.chmod()`

Table 2.6: Shell-Python compatibility

Function	Description
`os.walk()`	Iterate through the subfolders of a given directory.
`os.path.isdir()`	Return `True` if the input is a directory.
`os.path.isfile()`	Return `True` if the input is a file.
`os.path.join()`	Join several folder names or file names into one path.
`glob.glob()`	Return a list of file names that match a pattern.
`subprocess.call()`	Execute a shell command.
`subprocess.check_output()`	Execute a shell command and return its output as a string.

Table 2.7: Other useful Python functions for shell operations.

```
>>> import os
>>> from glob import glob

# Get the names of all Python files in the Python/ directory.
>>> glob("Python/*.py")
['Python/calc.py',
 'Python/count_files.py',
 'Python/mult.py',
 'Python/project.py']

# Get the names of all .jpg files in any subdirectory.
>> glob("**/*.jpg", recursive=True)
['Photos/IMG_1501.jpg',
 'Photos/IMG_1510.jpg',
 'Photos/IMG_1595.jpg',
 'Photos/img_1796.jpg',
 'Photos/img_1842.jpg',
 'Photos/img_1879.jpg',
 'Photos/img_1987.jpg',
 'Photos/IMG_2044.jpg',
 'Photos/IMG_2164.jpg',
 'Photos/IMG_2182.jpg',
 'Photos/IMG_2379.jpg',
 'Photos/IMG_2464.jpg',
```

```
 'Photos/IMG_2679.jpg',
 'Photos/IMG_2746.jpg']

# Walk through the directory, looking for .sh files.
>>> for directory, subdirectories, files in os.walk('.'):
...     for filename in files:
...         if filename.endswith(".sh"):
...             print(os.path.join(directory, filename))
...
./Scripts/five_secs.sh
./Scripts/script1.sh
./Scripts/script2.sh
./Scripts/script3.sh
./Scripts/ten_secs.sh
```

Problem 5. Write a Python function `grep()` that accepts the name of target string and a file pattern. Find all files in the current directory or its subdirectories that match the file pattern, then determine which ones contain the target string. For example, `grep("*.py", "range(")` should search Python files for the command `range(`.

Validate your function by comparing it to `grep -lR` in the shell.

The `subprocess` module allows Python to execute actual shell commands in the current working directory. Use `subprocess.call()` to run a Unix command, or `subprocess.check_output()` to run a Unix command and record its output.

```
$ cd Shell2/Scripts
$ python
>>> import subprocess
>>> subprocess.call(["ls", "-l"])
total 40
-rw-r--r--  1 username  groupname  20 Aug 26  2016 five_secs.sh
-rw-r--r--  1 username  groupname  21 Aug 26  2016 script1.sh
-rw-r--r--  1 username  groupname  21 Aug 26  2016 script2.sh
-rw-r--r--  1 username  groupname  21 Aug 26  2016 script3.sh
-rw-r--r--  1 username  groupname  21 Aug 26  2016 ten_secs.sh
0                               # decode() translates the result to a string.
>>> file_info = subprocess.check_output(["ls", "-l"]).decode()
>>> file_info.split('\n')
['total 40',
 '-rw-r--r--  1 username  groupname  20 Aug 26  2016 five_secs.sh',
 '-rw-r--r--  1 username  groupname  21 Aug 26  2016 script1.sh',
 '-rw-r--r--  1 username  groupname  21 Aug 26  2016 script2.sh',
 '-rw-r--r--  1 username  groupname  21 Aug 26  2016 script3.sh',
 '-rw-r--r--  1 username  groupname  21 Aug 26  2016 ten_secs.sh',
 '']
```

ACHTUNG!

Be extremely careful when creating a shell process from Python. If the commands depend on user input, the program is vulnerable to a *shell injection attack*. For example, consider the following function.

```
>>> def inspect_file(filename):
...     """Return information about the specified file from the shell."""
...     return subprocess.check_output(["ls", "-l", filename]).decode()
```

If `inspect_file()` is given the input `".; rm -rf /"`, then `ls -l .` is executed innocently, and then `rm -rf /` destroys the computer.[a] Be careful not to execute a shell command from within Python in a way that a malicious user could potentially take advantage of.

[a] See `https://en.wikipedia.org/wiki/Code_injection#Shell_injection` for more example attacks.

Problem 6. Write a Python function that accepts an integer n. Search the current directory and all subdirectories for the n largest files. Return a list of filenames, in order from largest to smallest.
(Hint: the shell commands `ls -s` and `du` show the file size.)

Downloading Files

The Unix shell has tools for downloading files from the internet. The most popular are `wget` and `curl`. At its most basic, `curl` is the more robust of the two while `wget` can download recursively.

When we want to download a single file, we just need the URL for the file we want to download. This works for PDF files, HTML files, and other content simply by providing the right URL.

```
$ wget https://github.com/Foundations-of-Applied-Mathematics/Data/blob/master/↩
    Volume1/dream.png
```

The following are also useful commands using `wget`.

```
# Download files from URLs listed in urls.txt
$ wget -i list_of_urls.txt

# Download in the background
$ wget -b URL

# Download something recursively
$ wget -r --no-parent URL
```

Problem 7. The file `urls.txt` in the `Documents/` directory contains a list of URLs. Download the files in this list using `wget` and move them to the `Photos/` directory.

When you finish this problem, archive and compress your entire `Shell2/` directory and save it as `ShellFinal.tar.gz`. Include the `-p` flag on `tar` to preserve the file permissions. (Hint: see the previous lab for a refresher on `tar`. See also `https://xkcd.com/1168/`.)

One Final Note

Though there are multiple Unix shells, one of the most popular is the *bash* shell. The bash shell is highly customizable. In your home directory, you will find a hidden file named `.bashrc`. All customization changes are saved in this file. If you are interested in customizing your shell, you can customize the prompt using the `PS1` environment variable. As you become more and more familiar with the Unix shell, you will come to find there are commands you run over and over again. You can save commands you use frequently using `alias`. If you would like more information on these and other ways to customize the shell, you can find many quality reference guides and tutorials on the internet.

Additional Material

sed and awk

`sed` and `awk` are two different scripting languages in their own right. Like Unix, these languages are easy to learn but difficult to master. It is very common to combine Unix commands and `sed` and `awk` commands. We will address the basics, but if you would like more information see <http://www.theunixschool.com/p/awk-sed.html>

Printing Specific Lines Using sed

We have already used the `head` and `tail` commands to print the beginning and end of a file respectively. What if we wanted to print lines 30 to 40, for example? We can accomplish this using `sed`. In the `Documents/` folder, you will find the `lines.txt` file. We will use this file for the following examples.

```
# Same output as head -n3
$ sed -n 1,3p lines.txt
line 1
line 2
line 3

# Same output as tail -n3
$ sed -n 3,5p lines.txt
line 3
line 4
line 5

# Print lines 2-4
$ sed -n 3,5p lines.txt
line 2
line 3
line 4

# Print lines 1,3,5
$ sed -n -e 1p -e 3p -e 5p lines.txt
line 1
line 3
line 5
```

Find and Replace Using sed

Using `sed`, we can also perform find and replace. We can perform this function on the output of another commmand or we can perform this function in place on other files. The basic syntax of this `sed` command is the following.

```
sed s/str1/str2/g
```

This command will replace every instance of `str1` with `str2`. More specific examples follow.

```
$ sed s/line/LINE/g lines.txt
LINE 1
LINE 2
LINE 3
LINE 4
LINE 5

# Notice the file didn't change at all
$ cat lines.txt
line 1
line 2
line 3
line 4
line 5

# To save the changes, add the -i flag
$ sed -i s/line/LINE/g lines.txt
$ cat lines.txt
LINE 1
LINE 2
LINE 3
LINE 4
LINE 5
```

Formatting output using awk

Earlier in this lab we mentioned `ls -l` and as we have seen, this outputs lots of information. Using `awk`, we can select which fields we wish to print. Suppose we only cared about the file name and the permissions. We can get this output by running the following command.

```
$ ls -l | awk ' {print $1, $9} '
```

Notice we pipe the output of `ls -l` to `awk`. When calling a command using `awk`, we use quotation marks. Note it is a common mistake to forget to add these quotation marks. Inside these quotation marks, commands always take the same format.

```
awk ' <options> {<actions>} '
```

In the remaining examples we will not be using any of the options, but we will address various actions. For those interested in learning what options are available see <http://www.theunixschool.com/p/awk-sed.html>.

In the `Documents/` directory, you will find a `people.txt` file that we will use for the following examples. In our first example, we use the `print` action. The `$1` and `$9` mean that we are going to print the first and ninth fields. Beyond specifying which fields we wish to print, we can also choose how many characters to allocate for each field.

```
# contents of people.txt
$ cat people.txt
male,John,23
female,Mary,31
female,Sally,37
male,Ted,19
male,Jeff,41
female,Cindy,25

# Change the field separator (FS) to "," at the beginning of execution (BEGIN)
# By printing each field individually proves we have successfully separated the↩
    fields
$ awk ' BEGIN{ FS = "," }; {print $1,$2,$3} ' < people.txt
male John 23
female Mary 31
female Sally 37
male Ted 19
male Jeff 41
female Cindy 25

# Format columns using printf so everything is in neat columns in order (gender↩
   ,age,name)
$ awk ' BEGIN{ FS = ","}; {printf "%-6s %2s %s\n", $1,$3,$2} ' < people.txt
male   23 John
female 31 Mary
female 37 Sally
male   19 Ted
male   41 Jeff
female 25 Cindy
```

The statement `"%-6s %2s %s\n"` formats the columns of the output. This says to set aside six characters left justied, then two characters right justified, then print the last field to its full length.

We have barely scratched the surface of what `awk` can do. Performing an internet search for "awk one-liners" will give you many additional examples of useful commands you can run using `awk`.

System Management

In this section, we will address some of the basics of system management. As an introduction, the commands in Table 2.8 are used to learn more about the computer system.

Command	Description
`passwd`	Change user password
`uname`	View operating system name
`uname -a`	Print all system information
`uname -m`	Print machine hardware
`w`	Show who is logged in and what they are doing
`whoami`	Print userID of current user

Table 2.8: Commands for system administration.

Secure Shell

Let's say you are working for a company with a file server. Hundreds of people need to be able to access the content of this machine, but how is that possible? Or say you have a script to run that requires some serious computing power. How are you going to be able to access your company's super computer to run your script? We do this through *Secure Shell* (SSH).

SSH is a network protocol encrypted using public-key cryptography. The system we are connecting *to* is commonly referred to as the *host* and the system we are connecting *from* is commonly referred to as the *client*. Once this connection is established, there is a secure tunnel through which commands and files can be exchanged between the client and host. To end a secure connection, type `exit`.

As a warning, you cannot normally SSH into a Windows machine. If you want to do this, search on the web for available options.

```
$ whoami    # use this to see what your current login is
client_username
$ ssh my_host_username@my_hostname

# You will then be prompted to enter the password for my_host_username

$ whoami    # use this to verify that you are logged into the host
my_host_username

$ hostname
my_hostname

$ exit
logout
Connection to my_host_name closed.
```

Now that you are logged in on the host computer, all the commands you execute are as though you were executing them on the host computer.

Secure Copy

When we want to copy files between the client and the host, we use the *secure copy* command, `scp`. The following commands are run when logged into the client computer.

```
# copy filename to the host's system at filepath
```

```
$ scp filename host_username@hostname:filepath

#copy a file found at filepath to the client's system as filename
$ scp host_username@hostname:filepath filename

# you will be prompted to enter host_username's password in both these ↩
    instances
```

Generating SSH Keys (Optional)

If there is a host that we access on a regular basis, typing in our password over and over again can get tedious. By setting up SSH keys, the host can identify if a client is a trusted user without needing to type in a password. If you are interested in experimenting with this setup, a Google search of "How to set up SSH keys" will lead you to many quality tutorials on how to do so.

3 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. Note that including a person's age in a database means that the data would quickly be outdated since people get older every year. A better choice would be to use birth year. Another table with the schema `(Name, Income)` would be $M \times 2$ if it included all M individuals.

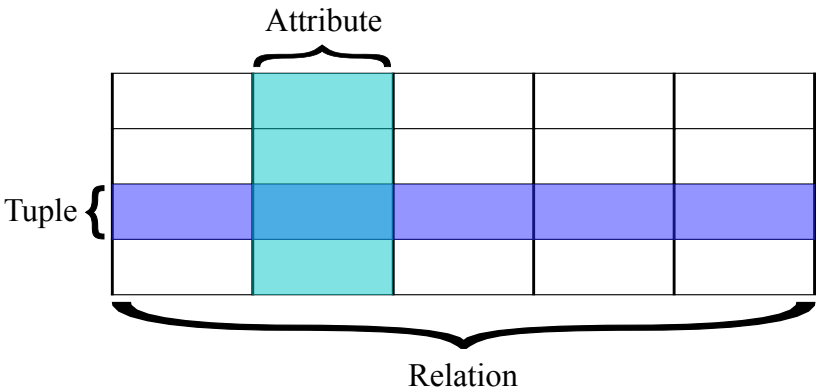


Figure 3.1: See `https://en.wikipedia.org/wiki/Relational_database`.

SQLite

The most common database management systems (DBMS) for relational databases are based on *Structured Query Language*, commonly called *SQL* (pronounced[1] "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.

[1] 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
None	NULL
int	INTEGER
float	REAL
str	TEXT
bytes	BLOB

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,[2] 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 3.1 for more table management commands.

[2]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	`CREATE TABLE <table> (<schema>);`
Delete a table	`DROP TABLE <table>;`
Delete a table if it exists	`DROP TABLE IF EXISTS <table>;`
Add a new column to a table	`ALTER TABLE <table> ADD <column> <dtype>`
Remove an existing column	`ALTER TABLE <table> DROP COLUMN <column>;`
Rename an existing column	`ALTER TABLE <table> ALTER COLUMN <column> <dtype>;`

Table 3.1: 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*.[a] 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.
```

[a] 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 3.2a–3.2d.

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 3.2: 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`[a] 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.)

[a] 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	`INSERT INTO table VALUES(<values>);`
Remove rows from a table	`DELETE FROM <table> WHERE <condition>;`
Change values in existing rows	`UPDATE <table> SET <column1>=<value1>, ... WHERE <condition>;`

Table 3.3: 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 3.4 to get the list of matching tuples.

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

Table 3.4: 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`

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;
```

4 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 4.1: Student database.

Joining Tables

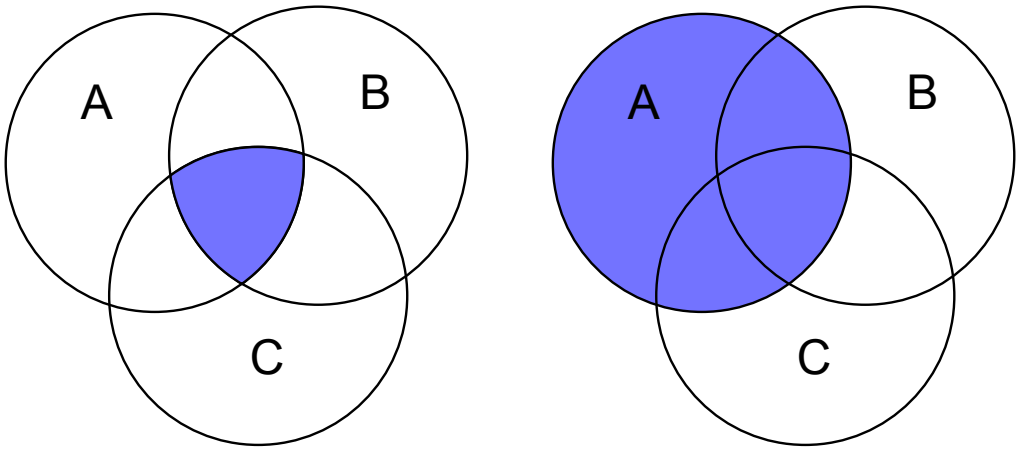
A *join* combines rows from different tables in a database based on common attributes. In other words, a join operation creates a new, temporary table containing data from 2 or more existing tables. Join commands in SQLite have the following general syntax.

```
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 4.1a.



(a) An inner join of A, B, and C.

(b) A left outer join of A with B and C.

Figure 4.1

For example, Table 4.1c (`StudentInfo`) and Table 4.1a (`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 4.2: 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 4.1a–4.1d, 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 4.2.

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 4.3: 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 4.3 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 are then combined by one of the *aggregate functions* `AVG()`, `MIN()`, `MAX()`, `SUM()`, or `COUNT()`. The following code groups the rows in Table 4.1d 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 4.2 or 4.3, 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.

5 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 object 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 `"^{@}(?)[%]{.}(*)[_]{&}$"`.

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

```
>>> 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.
```

There are also a variety of shortcuts that represent common character classes, listed in Table 5.1. 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 `[ \t\n\r\f\v]`.
`\S`	Matches any non-whitespace character; equivalent to `[^\s]`.
`\d`	Matches any decimal digit; equivalent to `[0-9]`.
`\D`	Matches any non-digit character; equivalent to `[^\d]`.
`\w`	Matches any alphanumeric character; equivalent to `[a-zA-Z0-9_]`.
`\W`	Matches any non-alphanumeric character; equivalent to `[^\w]`.

Table 5.1: 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 5.2: 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 5.3: 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
`match()`	Match a regular expression pattern to the beginning of a string.
`fullmatch()`	Match a regular expression pattern to all of a string.
`search()`	Search a string for the presence of a pattern.
`sub()`	Substitute occurrences of a pattern found in a string.
`subn()`	Same as sub, but also return the number of substitutions made.
`split()`	Split a string by the occurrences of a pattern.
`findall()`	Find all occurrences of a pattern in a string.
`finditer()`	Return an iterator yielding a match object for each match.

Table 5.4: 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
`re.DOTALL`	`.` matches any character at all, including the newline.
`re.IGNORECASE`	Perform case-insensitive matching.
`re.MULTILINE`	`^` matches the beginning of lines (after a newline) as well as the string; `$` matches the end of lines (before a newline) as well as the end of the string.

Table 5.5: 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. Return the string of code with colons in the correct places.

```
"""
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)
"""
```

Extracting Text with Regular Expressions

Regular expressions are useful for locating and extracting information that matches a certain format. The method `pattern.findall(string)` returns a list containing all non-overlapping matches of `pattern` found in `string`. The method scans the string from left to right and returns the matches in that order. If two matches overlap, the match that begins first is returned.

When at least one group, indicated by `()`, is present in the pattern, then only information contained in a group is returned. Each match is returned as a tuple containing the part of the string that matches each group in the pattern.

```
>>> pattern = re.compile("\w* fish")

# Without any groups, the entirety of each match is returned.
>>> pattern.findall("red fish, blue fish, one fish, two fish")
['red fish', 'blue fish', 'one fish', 'two fish']

# When a group is present, only information contained in a group is returned.
>>> pattern2 = re.compile("(\w*) (fish|dish)")
>>> pattern2.findall("red dish, blue dish, one fish, two fish")
[('red', 'dish'), ('blue', 'dish'), ('one', 'fish'), ('two', 'fish')]
```

If you wish to extract the characters that match some groups, but not others, you can choose to exclude a group from being returned using the syntax `(?:)`

```
>>> pattern = re.compile("(\w*) (?:fish|dish)")
>>> pattern.findall("red dish, blue dish, one fish, two fish")
['red', 'blue', 'one', 'two']
```

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.

Each contact name includes a first and last name. Some names have middle initials, in the form `Jane C. Doe`. Each birthday lists the month, then the day, and then the year, though the format varies from `1/1/11`, `1/01/2011`, etc. If century is not specified for birth year, as in `1/01/XX`, birth year is assumed to be `20XX`. Remember, not all information is listed for each contact.

Use regular expressions to extract the necessary 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 'regexp' 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 spreedsheet 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*.

6 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`.

```
{                                   # A family's info written in JSON format.
    "lastname": "Smith",            # The outer dictionary has two keys:
    "children": [                   # "lastname" and "children".
        {                           # The "children" key maps to a list of
            "name": "Timmy",        # two dictionaries, one for each of the
            "age": 8                # two children.
        },
        {
            "name": "Missy",
            "age": 5
        }
    ]
}
```

> **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.[a] 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.

[a] 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 3×3 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 O's go first."""
        self.board = [[' ']*3 for _ in range(3)]
        self.turn, self.winner = "O", None

    def move(self, i, j):
        """Mark an O 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 = "O" 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
`socket`	Create a new socket using the given address family, socket type and protocol number.
`bind`	Bind the socket to an address. The socket must not already be bound.
`listen`	Enable a server to accept connections.
`accept`	Accept a connection. Must be bound to an address and listening for connections.
`connect`	Connect to a remote socket at address.
`sendall`	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.
`recv`	Receive data from the socket. Must be given a buffer size; use 1024.
`close`	Mark the socket closed.

Table 6.1: 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/`.

7 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.[1] 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
```

[1] 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#  sioct: 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*.[a]

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`.[b]

[a] See `www.robotstxt.org/orig.html` and `en.wikipedia.org/wiki/Robots_exclusion_standard`.

[b] 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[2] 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>
```

[2] 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
`name`	The name of the tag
`attrs`	A dictionary of the attributes
`string`	The single string contained in the tag
`strings`	Generator for strings of children tags
`stripped_strings`	Generator for strings of children tags, stripping whitespace
`text`	Concatenation of strings from all children tags

Table 7.1: 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
`parent`	The parent tag
`parents`	Generator for the parent tags up to the top level
`next_sibling`	The tag immediately after to the current tag
`next_siblings`	Generator for sibling tags after the current tag
`previous_sibling`	The tag immediately before the current tag
`previous_siblings`	Generator for sibling tags before the current tag
`contents`	A list of the immediate children tags
`children`	Generator for immediate children tags
`descendants`	Generator for all children tags (recursively)

Table 7.2: 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, "attrs") 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.[a] 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.

[a] 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`

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>]

# Final 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.[a] 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.

[a] 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.[a] 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.

[a] See `http://www.federalreserve.gov/releases/lbr/20030930/default.htm`.

8 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.*

> **Note**
>
> For grading purposes, before returning a value in each function of this lab, write the value being returned to a `pickle` file. Name the file `ans1` if it contains the returned value of Problem 1, `ans2` for Problem 2, and so on, so that you have five pickle files saved. In order to create a pickle file, use the following code:
>
> ```
> >>> import pickle
> >>>
> >>> # Write the answer of Problem 1, "value", to a pickle file
> >>> with open("ans1", "wb") as fp:
> >>>     pickle.load(value, fp)
> ```

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 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`) that specifies which parts of the website are off-limits, and how often requests can be made according to the *robots exclusion standard*.[1]

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. [a]

[a] 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`.

The standard way to get the HTML source code of a website using Python is via the `requests` library.[2] 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
```

[1] See `www.robotstxt.org/orig.html` and `en.wikipedia.org/wiki/Robots_exclusion_standard`.

[2] 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/`.

```
>>> 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.

```
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_pod`, 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 £, 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. Return a list of lists where each list contains the assets of each bank.

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
`find_element_by_tag_name()`	The first tag with the given name
`find_element_by_name()`	The first tag with the specified `name` attribute
`find_element_by_class_name()`	The first tag with the given `class` attribute
`find_element_by_id()`	The first tag with the given `id` attribute
`find_element_by_link_text()`	The first tag with a matching `href` attribute
`find_element_by_partial_link_text()`	The first tag with a partially matching `href` attribute

Table 8.1: 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 website IMDB contains a variety of information on movies. Specifically, information on the top 10 box office movies of the week can be found at `https://www.imdb.com/chart/boxoffice`. Using BeaufiulSoup, Selenium, or both, return a list of the top 10 movies of the week and order the list according to the total grossing of the movies, from most money to the least.

Problem 5. 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 defaulting to `linkedin`. 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.

9 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.*

> **Note**
>
> This lab will be done using Colab Notebooks. These notebooks are similar to Jupyter Notebooks but run remotely on Google's servers. Open a Google Colab notebook by going to your Google Drive account and creating a new Colaboratory file. If making a Colaboratory file is not an option, download the application Colaboratory onto your Google Drive. Once opening a new Colab Notebook, upload the file `pandas1.ipynb`. To make the data files accessible, run the following at the top of the lab:
>
> ```
> >>> from google.colab import files
>
> >>> uploaded = files.upload()
> ```
>
> This will prompt you upload files for this notebook. For this lab, upload `budget.csv` and `crime_data.csv`.
>
> Once the lab is complete, delete BOTH lines of code used for uploading files (the import statement and the upload statement) and download as a `.py` file to your git repository. Push the newly made `pandas1.py` file.

Pandas Basics

Pandas is a python library used primarily to analyze data. It combines functionality of NumPy, MatPlotLib, and SQL to create a easy to understand library that allows for the manipulation of data in various ways. In this lab, we focus on the use of Pandas to analyze and manipulate data in ways similar to NumPy and SQL.

Pandas Data Structures

Series

The first pandas data structure is a `Series`. A `Series` is a one-dimensional array that can hold any datatype, similar to a `ndarray`. However, a `Series` has an `index` that gives a label to each entry. An `index` generally is used to label the data.

Typically a `Series` contains information about one feature of the data. For example, the data in a `Series` might show a class's grades on a test and the `Index` would indicate each student in the class. To initialize a `Series`, the first parameter is the data and the second is the index.

```
>>> import pandas as pd
>>>
>>> # Initialize Series of student grades
>>> math = pd.Series(np.random.randint(0,100,4), ['Mark', 'Barbara', 'Eleanor',↩
     'David'])
>>> english = pd.Series(np.random.randint(0,100,5), ['Mark', 'Barbara', 'David'↩
    , 'Greg', 'Lauren'])
```

DataFrame

The second key pandas data structure is a `DataFrame`. A `DataFrame` is a collection of multiple `Series`. It can be thought of as a 2-dimensional array, where each row is a separate datapoint and each column is a feature of the data. The rows are label with an `index` (as in a `Series`) and the columns are labelled in the attribute `columns`.

There are many different ways to initialize a `DataFrame`. One way to initialize a `DataFrame` is passing in a dictionary as the data of the `DataFrame`. The keys of the dictionary will become the labels in `columns` and the values are the `Series` associated with the label.

```
>>> # Create a DataFrame of student grades
>>> grades = pd.DataFrame({"Math": math, "English": english}
>>> grades
          Math  English
Barbara   52.0     73.0
David     10.0     39.0
Eleanor   35.0      NaN
Greg       NaN     26.0
Lauren     NaN     99.0
Mark      81.0     68.0
```

Notice that `pd.DataFrame` automatically lines up data from both `Series` that have the same index. If the data only appears in one of the `Series`, the entry for the second `Series` is `NaN`.

We can also initialize a `DataFrame` with a NumPy array. In this way, the data is passed in as a 2-dimensional NumPy array, while the column labels and index are passed in as parameters. The first column label goes with the first column of the array, the second with the second, etc. The same holds for the index.

```
>>> import numpy as np
>>> # Initialize DataFrame with NumPy array
```

```
>>> data = np.array([[52.0, 73.0], [10.0, 39.0], [35.0, np.nan], [np.nan, ↩
    26.0], [np.nan, 99.0], [81.0, 68.0]])
>>> grades = pd.DataFrame(data, columns = ['Math', 'English'], index = ['↩
    Barbara', 'David', 'Eleanor', 'Greg', 'Lauren', 'Mark'])
>>> grades
          Math  English
Barbara   52.0     73.0
David     10.0     39.0
Eleanor   35.0     NaN
Greg       NaN     26.0
Lauren     NaN     99.0
Mark      81.0     68.0
```

A `DataFrame` can also be viewed as a NumPy array using the attribute `values`.

```
>>> # View the DataFrame as a NumPy array
>>> grades.values
array([[ 52.,  73.],
       [ 10.,  39.],
       [ 35.,  nan],
       [ nan,  26.],
       [ nan,  99.],
       [ 81.,  68.]])
```

Problem 1. Write a function `random_dataframe()` that accepts a dictionary `d` which defaults to `None`. If a dictionary is passed in, initialize a Pandas `DataFrame`. Return a tuple of the attributes `index`, `columns`, and `values` of the `DataFrame`.

If a dictionary is not passed in, generate random data as a `ndarray` and initialize a `DataFrame`. The columns of the `DataFrame` should be the letters `'A'` through `'E'`. The index of the `DataFrame` should be the roman numerals 1-6. Return a tuple of the attributes `index`, `columns`, and `values` of the `DataFrame`.

(Hint: What should the dimension of the data be if no dictionary is passed in?)

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

Table 9.1: 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 (0 indexed) in the CSV file that contains the column names.
- `index_col`: The column (0 indexed) 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.

Data Manipulation

Accessing Data

While array slicing can be used to access data in a `DataFrame`, it is always preferable to use the `loc` and `iloc` 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 `iloc` selects them based on their integer position. When using these indexers, the first and second arguments refer to the rows and columns, respectively, just as array slicing.

```
>>> grades
          Math  English
Barbara   52.0     73.0
David     10.0     39.0
Eleanor   35.0     NaN
Greg       NaN     26.0
Lauren     NaN     99.0
Mark      81.0     68.0

>>> # Use loc to select the Math scores of David and Greg
>>> grades.loc[['David', 'Greg'],'Math']
David    10.0
Greg      NaN
Name: Math, dtype: float64

>>> # Use iloc to select the Math scores of David and Greg
>>> grades.iloc[[1,3], 0]
David    10.0
Greg      NaN
```

An entire column of a `DataFrame` can be accessed using simple square brackets and the name of the column. In addition, to create a new column or reset the values of an entire column, simply call this column in the same fashion and set the value.

```
>>> # Set new History column with array of random values
>>> grades['History'] = np.random.randint(0,100,6)
>>> grades['History']
Barbara     4
David      92
Eleanor    25
Greg       79
Lauren     82
Mark       27
Name: History, dtype: int64

>>> # Reset the column such that everyone has a 100
>>> grades['History'] = 100.0
>>> grades
         Math  English  History
Barbara  52.0     73.0    100.0
David    10.0     39.0    100.0
Eleanor  35.0      NaN    100.0
Greg      NaN     26.0    100.0
Lauren    NaN     99.0    100.0
Mark     81.0     68.0    100.0
```

Often datasets can be very large and difficult to visualize. Pandas offers various methods to make the data easier to visualize. The methods `head` and `tail` will show the first or last n data points, respectively, where n defaults to 5. The method `sample` will draw n random entry of the dataset, where n defaults to 1.

```
>>> # Use head to see the first n rows
>>> grades.head(n=2)
         Math  English  History
Barbara  52.0     73.0    100.0
David    10.0     39.0    100.0

>>> # Use sample to sample a random entry
>>> grades.sample()
        Math  English  History
Lauren   NaN     99.0    100.0
```

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

```
>>> # Re-order columns
>>> grades.reindex(columns['English','Math','History'])
         English  Math  History
```

```
Barbara     73.0  52.0    100.0
David       39.0  10.0    100.0
Eleanor      NaN  35.0    100.0
Greg        26.0   NaN    100.0
Lauren      99.0   NaN    100.0
Mark        68.0  81.0    100.0

>>> # Sort descending according to Math grades
>>> grades.sort_values('Math', ascending=False)
         Math  English  History
Mark     81.0     68.0    100.0
Barbara  52.0     73.0    100.0
Eleanor  35.0      NaN    100.0
David    10.0     39.0    100.0
Greg      NaN     26.0    100.0
Lauren    NaN     99.0    100.0
```

Other methods used for manipulating `DataFrame` and `Series` panda structures can be found in Table 9.2.

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

Table 9.2: Methods for managing or modifying data in a pandas `Series` or `DataFrame`.

Problem 2. The file `budget.csv` contains the budget of a college student over the course of 4 years. Write a function `prob2()` reads in `budget.csv` as a `DataFrame`. Perform the following operations:

1. Reindex the columns such that amount spent on food is the first column and all other columns maintain the same ordering.
2. Sort the `DataFrame` in descending order based on how much money was spent on `Groceries`
3. Reset all values in the `'Rent'` column to `800.0`
4. Reset all values in the first 5 data points to `0.0`

Return the values of the updated `DataFrame` as a NumPy array.

Basic Data Manipulation

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

```
>>> # Sum history and english grades of all students
>>> grades['English'] + grades['History']
Barbara    173.0
David      139.0
Eleanor      NaN
Greg       126.0
Lauren     199.0
Mark       168.0
dtype: float64

>>> # Double all Math grades
>>>  grades['Math']*2
Barbara    104.0
David       20.0
Eleanor     70.0
Greg         NaN
Lauren       NaN
Mark       162.0
Name: Math, dtype: float64
```

In addition to arithmetic, `Series` have a variety of other methods similar to NumPy arrays. A collection of these methods is found in Table 9.3.

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

Table 9.3: Numerical methods of the `Series` and `DataFrame` pandas classes.

Basic Statistical Functions

The pandas library allows us to easily calculate basic summary statistics of our data, useful when we want a quick description of the data. The `describe()` function outputs several such summary statistics for each column in a `DataFrame`:

```
>>> # Use describe to better understand the data
>>> grades.describe()
            Math   English  History
count   4.000000   5.00000      6.0
mean   44.500000  61.00000    100.0
std    29.827281  28.92231      0.0
min    10.000000  26.00000    100.0
25%    28.750000  39.00000    100.0
50%    43.500000  68.00000    100.0
75%    59.250000  73.00000    100.0
max    81.000000  99.00000    100.0
```

Functions for calculating means and variances, the covariance and correlation matrices, and other basic statistics are also available.

```
>>> # Find the average grade for each student
>>> grades.mean(axis=1)
Barbara    75.000000
David      49.666667
Eleanor    67.500000
Greg       63.000000
Lauren     99.500000
Mark       83.000000
dtype: float64

>>> # Solve for the unbiased variance between subjects
>>> grades.cov()
               Math  English  History
Math     889.666667    557.0      0.0
English  557.000000    836.5      0.0
History    0.000000      0.0      0.0

>>> # Give correlation matrix between subjects
>>> grades.corr()
            Math  English  History
Math     1.00000  0.84996      NaN
English  0.84996  1.00000      NaN
History      NaN      NaN      NaN
```

The method `rank` gives a ranking based on methods such as average, minimum, and maximum. This method defaults ranking in ascending order (the least will be ranked 1 and the greatest will be ranked the highest number).

```
>>> # Rank each student's performance based on their highest grade in any class↩
     in descending order
>>> grades.rank(axis=1,method='max',ascending=False)
         Math  English  History
Barbara   3.0      2.0      1.0
David     3.0      2.0      1.0
Eleanor   2.0      NaN      1.0
Greg      NaN      2.0      1.0
Lauren    NaN      2.0      1.0
Mark      2.0      3.0      1.0
```

These methods can be very effective in interpreting data. For example, the `rank` example above shows use that Barbara does best in History, then English and then Math.

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:

```
>>> # Grades with all NaN values dropped
>>> grades.dropna()
         Math  English  History
Barbara  52.0     73.0    100.0
David    10.0     39.0    100.0
Mark     81.0     68.0    100.0
```

This is not always the desired behavior, however. Missing data could actually correspond to some default value, such as zero. For example, in the budget dataset, filling `NaN` value with 0 indicates that no money was spent on that item. In the grade dataset, we can replace all instances of `NaN` with a specified value:

```
>>> # fill missing data with 50.0
>>> grades.fillna(50.0)
         Math  English  History
Barbara  52.0     73.0    100.0
David    10.0     39.0    100.0
Eleanor  35.0     50.0    100.0
Greg     50.0     26.0    100.0
Lauren   50.0     99.0    100.0
Mark     81.0     68.0    100.0
```

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.

Achtung!

Always consider missing data carefully when analyzing a dataset. It may not always be helpful to drop the data or fill it in with a random number. Consider filling the data with the mean of surrounding data or the mean of the feature in question. Overall, the choice for how to fill missing data should make sense with the dataset.

Problem 3. Write a function `prob3()` that uses `budget.csv` to answer the questions "Which category affects living expenses the most? Which affects other expenses the most? How much is generally spent in these two categories?". Use the functions above to manipulate the data to perform the following manipulations:

1. Fill all `NaN` values with `0.0`.
2. Create two new columns, `'Living Expenses'` and `'Other'`. Sum the columns `'Rent'`, `'Groceries'`, `'Gas'` and `'Utilities'` and set as the value of `'Living Expenses'`. Sum the columns `'Dining Out'`, `'Out With Friends'` and `'Netflix'` and set as the value of `'Other'`.
3. Identify which column correlates most with `'Living Expenses'` and which correlates most with `'Other'`. This can indicate which columns in the budget affects the overarching categories the most.

Return the mean of each the two columns found in 3 as a tuple. The first mean should be of the column corresponding to `'Living Expenses'` and the second to `'Other'`.

SQL Operations in pandas

`DataFrames` are tabular data structures bearing an obvious resemblance to a typical relational database table. SQL is the standard for working with relational databases; however, pandas can accomplish many of the same tasks as SQL. The SQL-like functionality of pandas is one of its biggest advantages, eliminating 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)
```

SQL SELECT statements can be done by column indexing. WHERE statements can be included by adding masks (just like in a NumPy array). The method `isin()` can also provide a useful WHERE statement. This method accepts a list, dictionary, or `Series` containing possible values of the `DataFrame` or `Series`. When called upon, it returns a `Series` of booleans, indicating whether an entry contained a value in the parameter pass into `isin()`.

```
>>> # 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'
>>> mask = otherInfo['Financial_Aid'] == 'y'
>>> otherInfomask][['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
```

Next, let's look at JOIN statements. In pandas, this is done with the `merge` function. `merge` takes the two `DataFrame` objects to join as parameters, 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 4. 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'.

Create a new column `Rate` which contains the crime rate for each year. Using panda commands, find the number of murders in the years where the crime rate was greater than 5% and the number of `Violent` was more than on average. Return an array containing the number of murders in these years.

(Hint: To do `AND` statements, use two masks. Use `values` attribute to create array.)

Problem 5. Answer the following questions using the file `crime_data.csv` and the pandas methods learned in this lab. The answer of each question should be saved as indicated. Return the answers to each question as a tuple (i.e. `(answer_1,answer_2,answer_3)`).

1. Identify the three crimes that have a mean over 1,500,000. Of these three crimes, which two are very correlated? Which of these two crimes has a greater maximum value? Save the title of this column as a variable to return as the answer.
2. Examine the data since 2000. Sort this data (in ascending order) according to number of murders. SELECT Aggravated Assault WHERE Aggravated Assault is greather than 850,000. Save the reordered and SQL queried `DataFrame` as a NumPy array to return as the answer.
3. What decade had the most crime? In this decade, which crime was committed the most? What percentage of the total crime that year was it? Save this value as a float.

10 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 the data.*

> **Note**
>
> This lab will be done using Colab Notebooks. These notebooks are similar to Jupyter Notebooks but run remotely on Google's servers. Open a Google Colab notebook by going to your Google Drive account and creating a new Colaboratory file. If making a Colaboratory file is not an option, download the application Colaboratory onto your Google Drive. Once opening a new Colab Notebook, upload the file `pandas2.ipynb`. To make the data files accessible, run the following at the top of the lab:
>
> ```
> >>> from google.colab import files
>
> >>> uploaded = files.upload()
> ```
>
> This will prompt you upload files for this notebook. For this lab, upload `crime_data.csv` and `college.csv`.
>
> Once the lab is complete, delete BOTH lines of code used for uploading files (the import statement and the upload statement) and download as a `.py` file to your git repository. Push the newly made `pandas2.py` file.

Overview of Plotting Tools

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

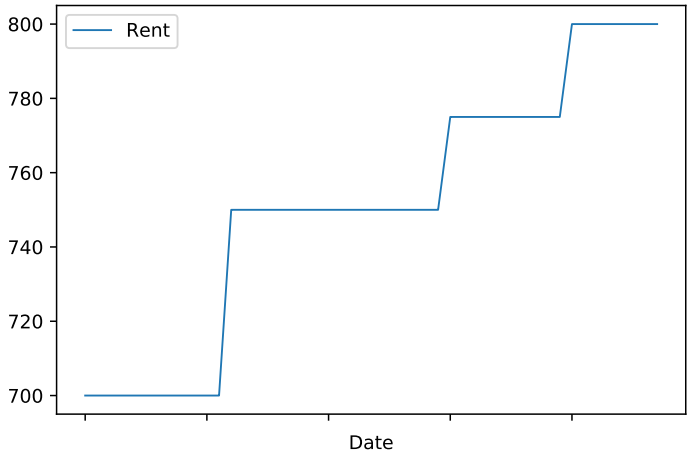
Plot Type	`plot()` ID	Uses and Advantages
Line plot	`"line"`	Show trends ordered in data; easy to compare multiple data sets
Scatter plot	`"scatter"`	Compare exactly two data sets, independent of ordering
Bar plot	`"bar"`, `"barh"`	Compare categorical or sequential data
Histogram	`"hist"`	Show frequencies 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 10.1: Types of plots in pandas. 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` 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

>>> budget = pd.read_csv("new_budget.csv", index_col="Year")
>>> budget.plot(y="Rent")  # Plot rent against the index (date).
```



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

```
>>> plt.plot(budget.index, budget['Rent'], label='Rent')
>>> plt.xlabel(budget.index.name)
>>> plt.xlim(min(budget.index), max(budget.index))
>>> plt.legend(loc='best')
```

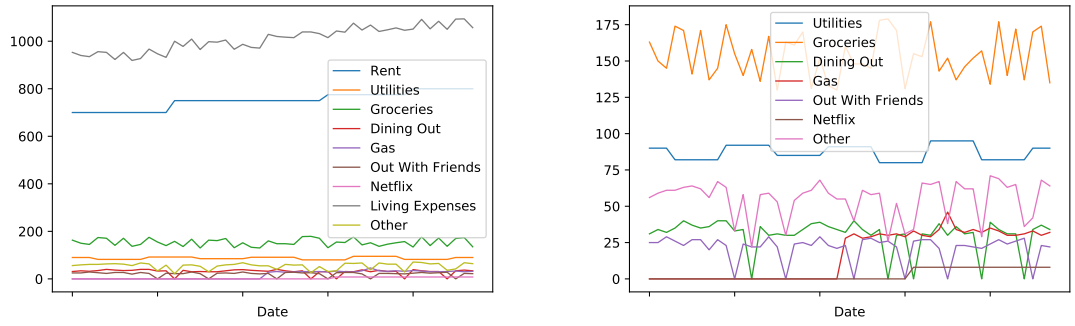
The `plot()` method also takes in many keyword arguments for matplotlib plotting and annotation functions. For example, setting `legend=False` disables the legend, providing a value for `title` sets the figure title, `grid=True` turns a grid on, and so on. For more customizations, see `https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.plot.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. By default, the `plot()` method attempts to plot **every** `Series` (column) in a `DataFrame`. This is especially useful with sequential data, like the budget data set.

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

>>> # Plot all columns together except for 'Living Expenses' and 'Rent'.
>>> budget.drop(["Living Expenses", "Rent"], axis=1).plot(linewidth=1)
```



(a) All columns of the budget data set on the same figure, using the index as the x-axis.

(b) All columns of the budget data set except `"Living Expenses"` and `"Rent"`.

Figure 10.1

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 budget data set has 9 columns, so the resulting figure, Figure 10.1a, is fairly cluttered.

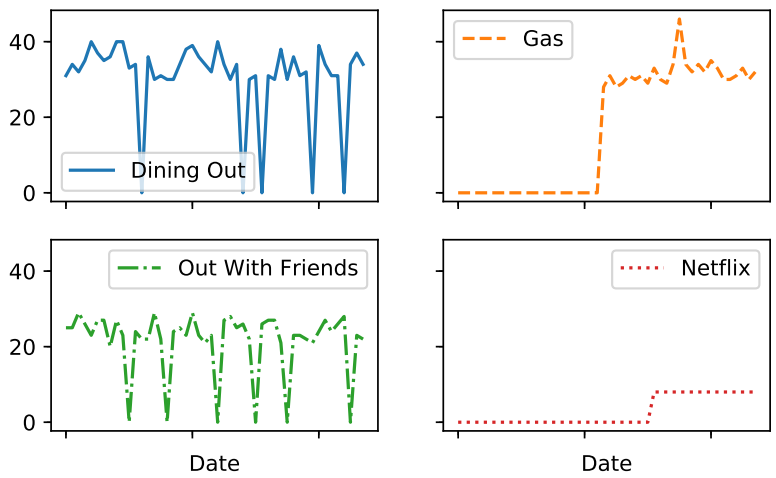
One way to declutter a visualization is to examine less data. Notice that `'Living Expenses'` has values much bigger than the other columns. Dropping this column, as well as `'Rent'`, gives a better overview of the data, shown in Figure 10.1b.

ACHTUNG!

Often plotting all data at once is unwise because columns have **different units of measure**. Be careful not to plot parts of a data set together if those parts do not 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.

```
>>> budget.plot(y=['Dining Out','Gas','Out With Friends', 'Netflix'],
...                 subplots=True, layout=(2,2), sharey=True,
...                 style=['-','--','-.',':'])
```



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

```
>>> # Plot three histograms together.
>>> budget.plot(kind='hist',y=['Gas','Dining Out','Out With Friends'],
...              alpha=.7,bins=10)

>>> # Plot three histograms, stacking one on top of the other.
>>> budget.plot(kind='hist',y=['Gas','Dining Out','Out With Friends'],
...                 bins=10,stacked=True)
```

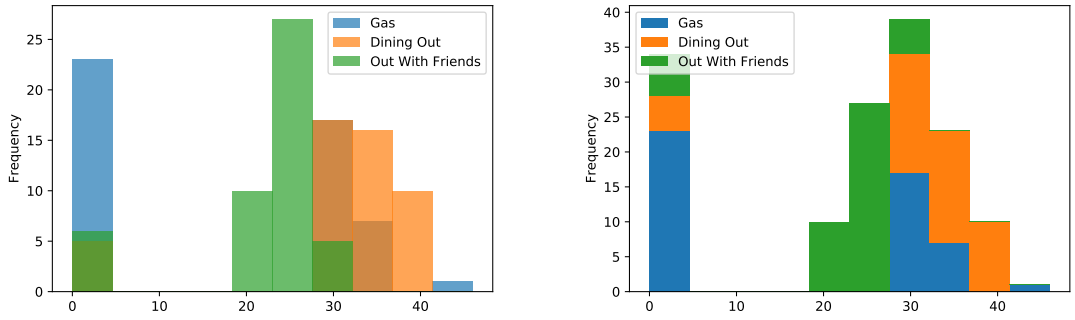


Figure 10.2: Two examples of histograms that are difficult to understand because multiple columns are plotted.

Thus, histograms are good for examining the distribution of a **single** column in a data set. For histograms, use the `hist()` method of the `DataFrame` instead of the `plot()` method. Specify the number of bins with the `bins` parameter. Choose a number of bins that accurately represents the data; the wrong number of bins can create a misleading or uninformative visualization.

```
>>> budget[["Dining Out","Gas","Other","Out With Friends"]].hist(grid=False, ↩
    bins=10)
```

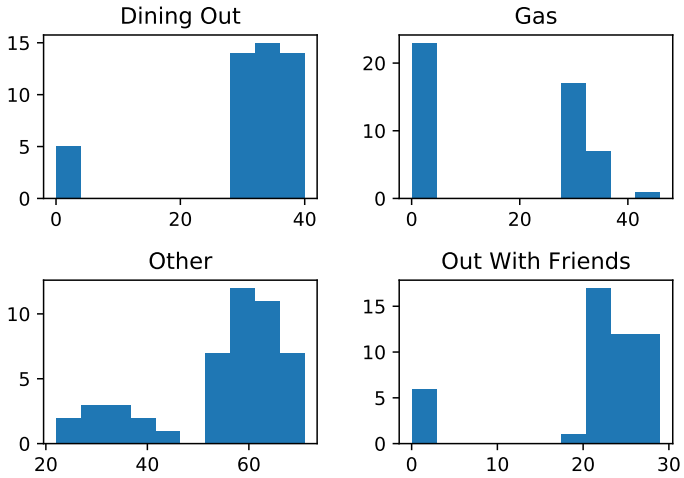


Figure 10.3: Histograms of `"Dining Out"`, `"Gas"`, `"Other"`, and "Out With Friends".

Problem 1. Create 2 visualizations for the data in `crime_data.csv`. Make one of the visualizations a histogram. The visualizations should be well labeled and easy to understand. Include a short description of your plots as a caption.

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 10.1b suggests that the `"Dining Out"` and `"Out With Friends"` 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 `x` and `y` columns as arguments.

```
>>> # Plot 'Dining Out' and 'Out With Friends' as lines against the index.
>>> budget.plot(y=["Dining Out", "Out With Friends"])

>>> # Make a scatter plot of 'Dining Out' against 'Out With Friends'
>>> budget.plot(kind="scatter", x="Dining Out", y="Out With Friends",
...                 alpha=.8)
```

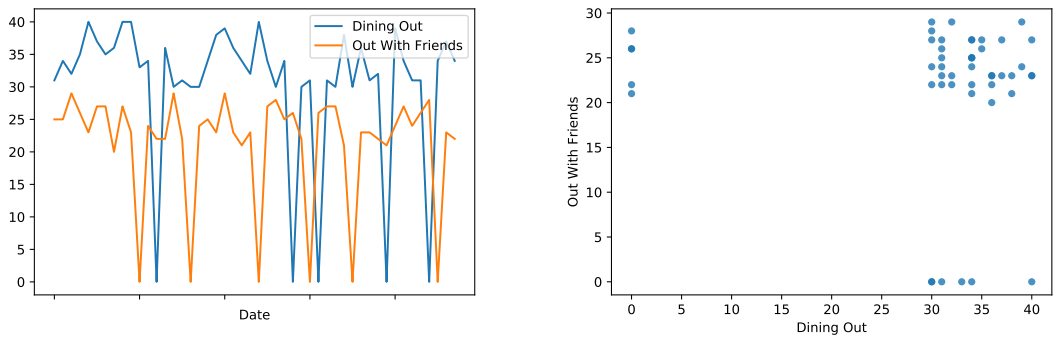


Figure 10.4: Correlations between `"Dining Out"` and `"Out With Friends"`.

The first plot shows us that more money is spent on dining out than being out with friends overall. However, both categories stay in the same range for most of the data. This is confirmed in the scatter plot by the block in the upper right corner, indicating the common range spent on dining out and being out with friends.

Achtung!

When 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 from Problem 1 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.

Another useful visualization used to understand correlations in 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 quick method for an initial analysis of the correlation between different columns.

```
>>> pd.plotting.scatter_matrix(budget[['Living Expenses','Other']])
```

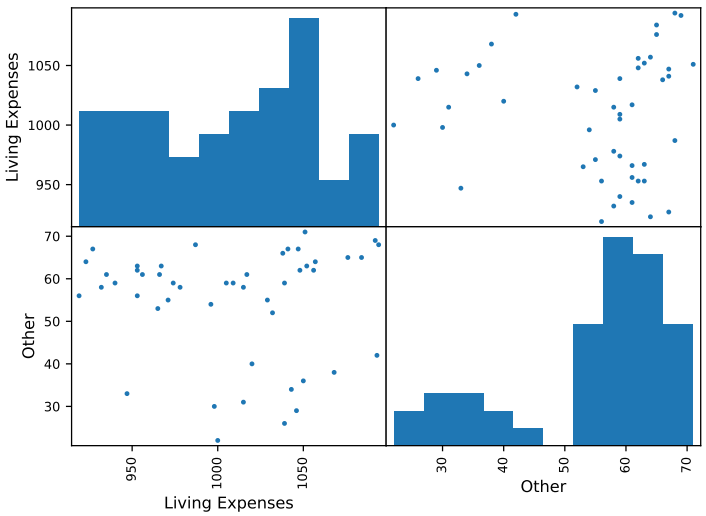


Figure 10.5: Scatter matrix comparing `"Living Expenses"` and `"Other"`.

Bar Graphs

Different types of graphs help to identify different patterns. Note that the data set `budget` gives monthly expenses. It may be beneficial to look at one specific month. Bar graphs are a good way to compare small portions of the data set.

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 all data for the last month in the budget
>>> budget.iloc[-1,:].plot(kind='barh')
>>> plt.tight_layout()

>>> # Plot all data for the last month without 'Rent' and 'Living Expenses'
>>> budget.drop(['Rent','Living Expenses'],axis=1).iloc[-1,:].plot(kind='barh')
>>> plt.tight_layout()
```

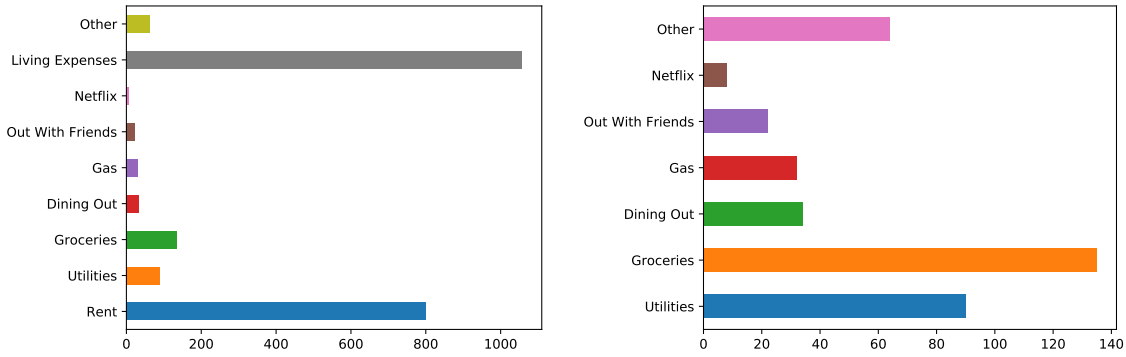


Figure 10.6: Bar graphs showing expenses paid in the last month of `budget`.

Problem 2. Using the crime data from the previous problem, identify if a trend exists between `Forcible Rape` and the following variables:

1. `Violent`
2. `Burglary`
3. `Aggravated Assault`

Make sure each graph is clearly labelled and readable. Include a caption explaining whether there is a visual trend between the variables.

Distributional Visualizations

While histograms are good at displaying the distributions for 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 four columns.
>>> budget.plot(kind="box", y=["Gas","Dining Out","Out With Friends","Other"])

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

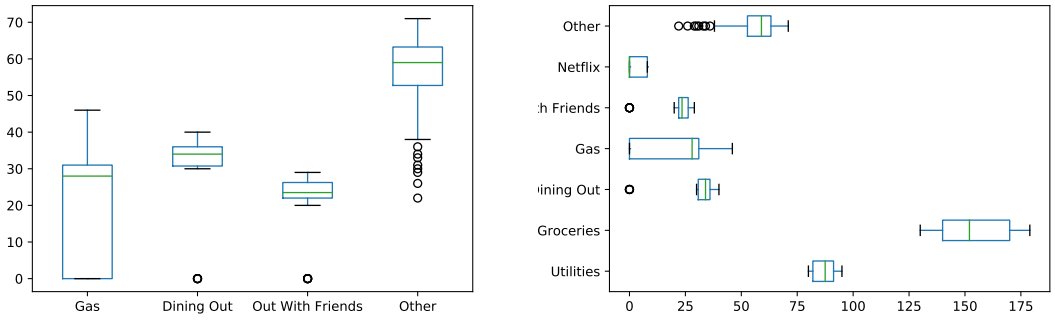


Figure 10.7: Vertical and horizontal box plots of `budget` dataset.

Hexbin Plots

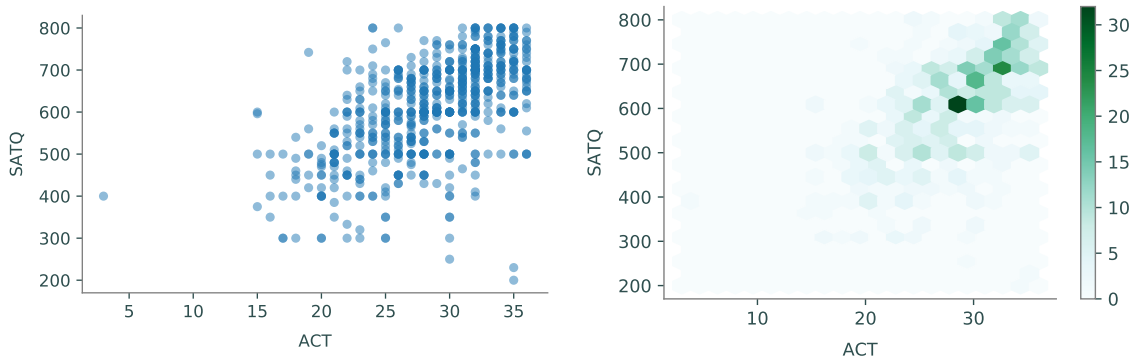
A scatter plot is essentially a plot of samples from the joint distribution of two columns. However, scatter plots can be uninformative for large data sets when the points in a scatter plot are closely clustered. *Hexbin plots* solve this problem by plotting point density in hexagonal bins—essentially 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 10.8a, is highly cluttered, even though the points have some transparency. A hexbin plot of the same data, Figure 10.8b, 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 10.8: Scatter plots and hexbin plot of SAT and ACT scores.

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. Use `crime_data.csv` to display the following distributions.

1. The distributions of `Burglary`, `Violent`, and `Vehicle Theft` across all crimes,
2. The distributions of `Vehicle Thefts` over the values of `Robbery`.

As usual, all plots should be labeled and easy to read.

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 10.9. 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, nor does it tell what `cons` is. There is no title, and the source of the data is unknown.

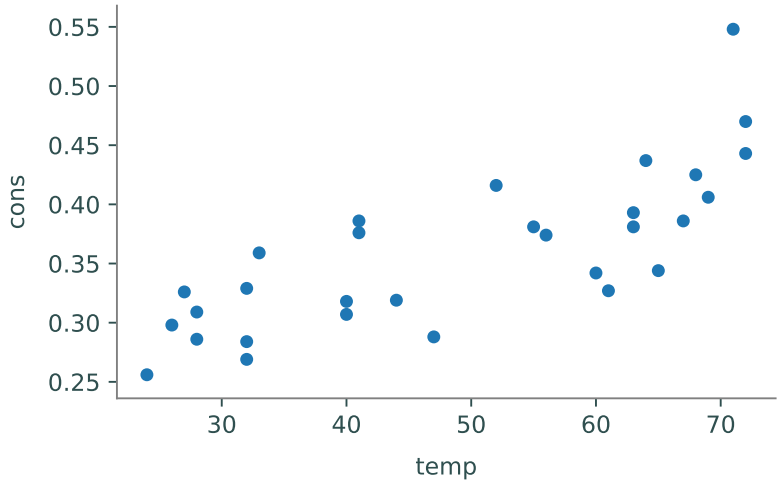


Figure 10.9: Non-specific data.

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 10.9. 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 10.9. Examining the source of the dataset can give important details to create better plots. When plotting data, make sure to understand what the variable names represent and where the data was taken from. Use this information to create a more effective plot.

The ice cream data used in Figure 10.9 is better understood with 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 (Farenheit)")
>>> plt.ylabel("Consumption per head (pints)")
```

To 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 easily replaced by a caption attached to the figure. 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 10.10.

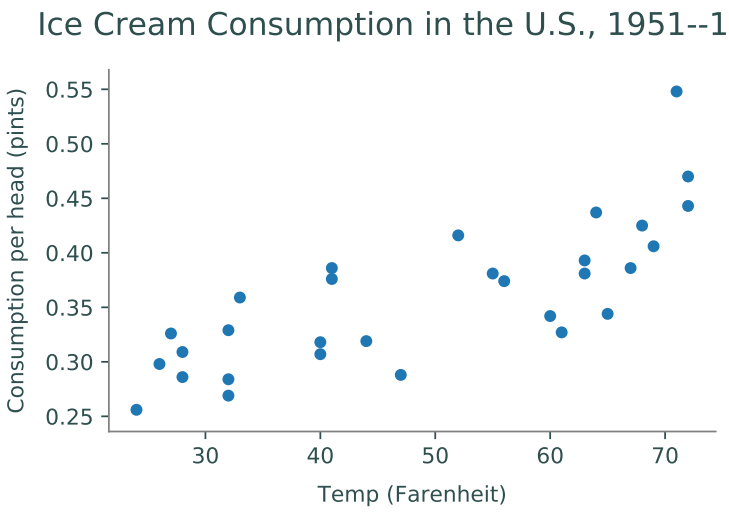


Figure 10.10: Source: Hildreth, C. and J. Lu (1960) *Demand relations with autocorrelated disturbances*, Technical Bulletin No 2765, Michigan State University.

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 dataset `college.csv` contains information from 1995 on universities in the United States. To access information on variable names, go to `https://cran.r-project.org/web/packages/ISLR/ISLR.pdf`. Create 3 plots that compare variables or universities. These plots should answer questions about the data, e.g. what is the distribution of graduation rates or do schools with more PhD students also take more students from the Top 10 percent of their high school class. These plots should be easy to understand, have clear variable names, and citations.

11 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.*

Note

This lab will be done using Colab Notebooks. These notebooks are similar to Jupyter Notebooks but run remotely on Google's servers. Open a Google Colab notebook by going to your Google Drive account and creating a new Colaboratory file. If making a Colaboratory file is not an option, download the application Colaboratory onto your Google Drive. Once opening a new Colab Notebook, upload the file `pandas3.ipynb`. To make the data files accessible, run the following at the top of the lab:

```
>>> from google.colab import files

>>> uploaded = files.upload()
```

This will prompt you upload files for this notebook. For this lab, upload `college.csv` and `ohio_1999.csv`.

Once the lab is complete, delete BOTH lines of code used for uploading files (the import statement and the upload statement) and download as a `.py` file to your git repository. Push the newly made `pandas3.py` file.

Groupby

The file `mammal_sleep.csv`[1] 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  carni     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  carni     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")
```

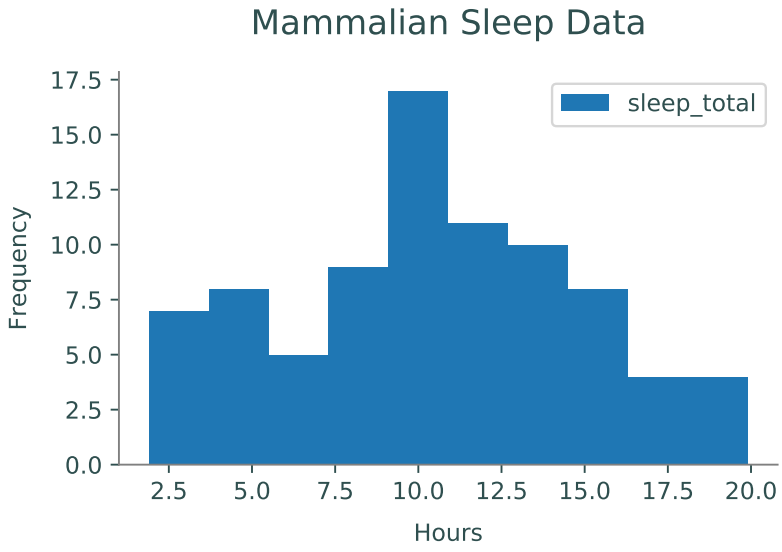


Figure 11.1: `"sleep_total"` frequencies from the mammalian sleep data set.

[1] 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"`.

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
```

As shown above, `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 easily it compares 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  carni  Cetacea          2.7
59       Common porpoise       Phocoena  carni  Cetacea          5.6
79  Bottle-nosed dolphin       Tursiops  carni  Cetacea          5.2
```

Problem 1. Read in the data `college.csv` containing information on various United States universities in 1995. To access information on variable names, go to `https://cran.r-project.org/web/packages/ISLR/ISLR.pdf`. Use a `groupby` object to group the colleges by private and public universities. Read in the data as a `DataFrame` object and use `groupby` and `describe` to examine the following columns by group:

1. Student to Faculty Ratio,
2. How many students from the top 25% of their high school class,
3. How many students from the top 10% of their high school class.

Determine whether private or public universities have a higher mean for each of these columns. For the type of university with the higher mean, save the values of the describe function on said column as an array using `.values`. Return a tuple with these arrays in the order described above.

For example, if I were comparing whether the number of professors with PhDs was higher at private or public universities, I would return the following array:

```
array([212., 76.83490566, 12.31752531, 33., 71., 78.5 , 86., 103.])
```

Visualizing Groups

There are a few ways that `groupby()` can simplify the process of visualizing groups of data. First of all, `groupby()` makes it easy to visualize one group at a time using the `plot` method. The following visualization improves on Figure 11.1 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")
```

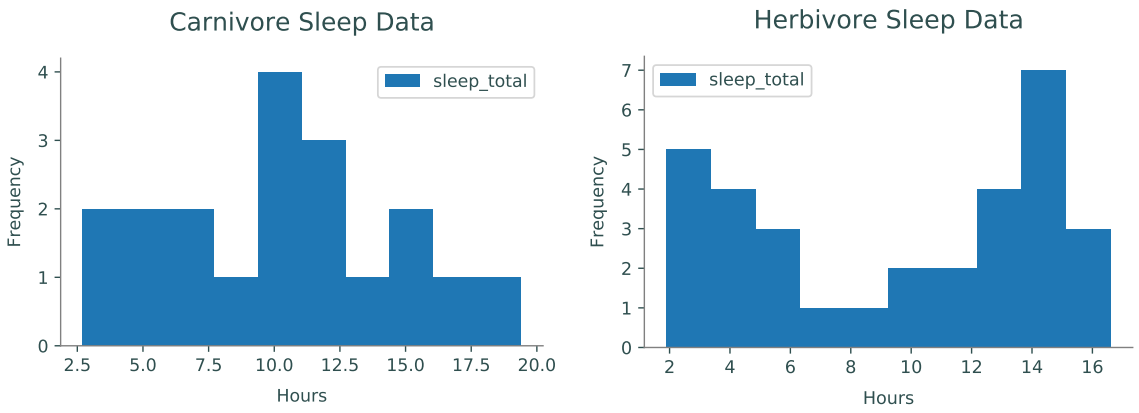
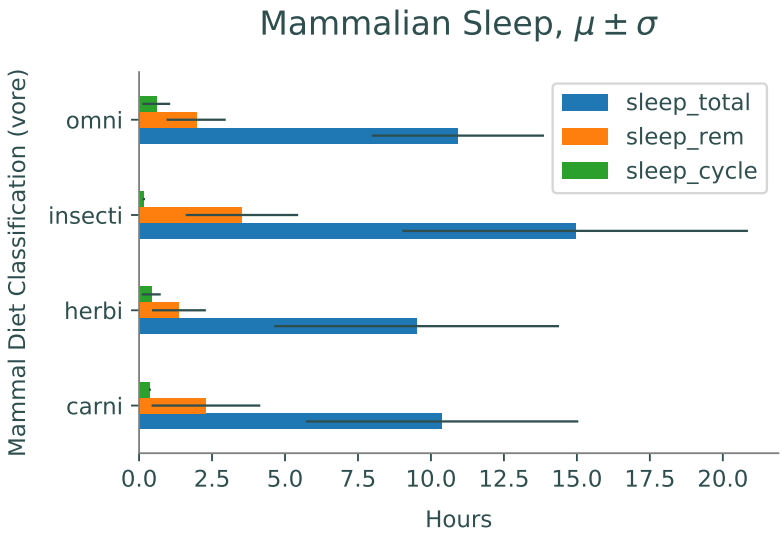


Figure 11.2: `"sleep_total"` histograms for two groups in the mammalian sleep data set.

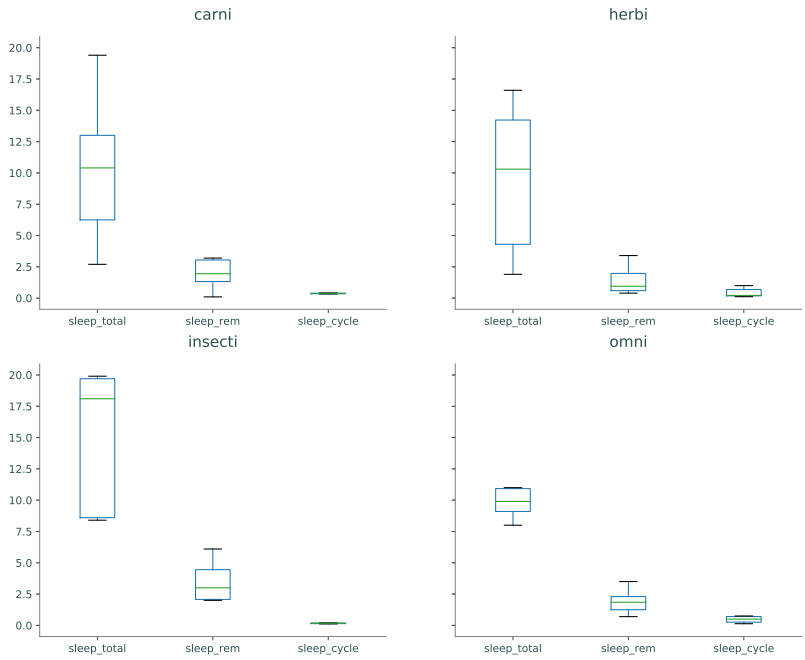
The statistical summaries from the `GroupBy` object's `mean()`, `std()`, or `describe()` methods also lend themselves well to certain visualizations for comparing groups.

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



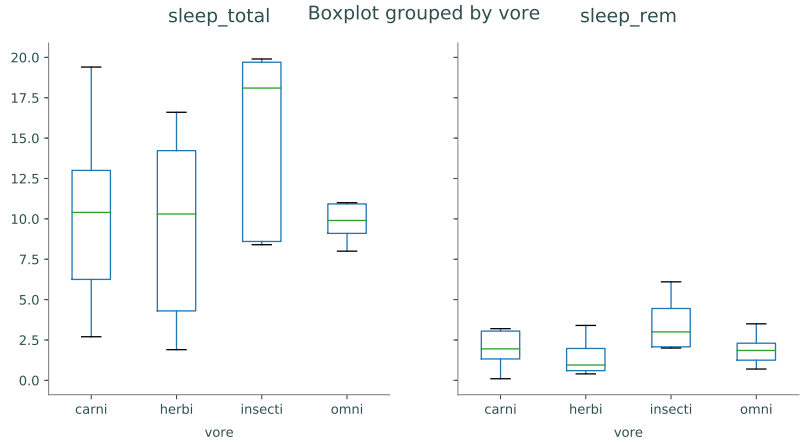
Box plots are well suited for comparing similar distributions. The `boxplot()` method of the `GroupBy` class creates one subplot **per group**, plotting each of the columns as a box plot.

```
# 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 2. Create visualizations that give relevant information answering the following questions (using `college.csv`):

1. How do the number of applicants, number of accepted students, and number of enrolled students compare between private and public universities?
2. How wide is the range of money spent on room and board at both private and public universities?

Pivot Tables

One of the downfalls 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. An example in the previous dataset would be 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.

Consider the Titanic data set found in `titanic.csv`[2]. 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(),)

>>> 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
```

Note

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

```
>>> titanic.groupby(["Sex", "Pclass"])["Survived"].mean().unstack()
Pclass    1.0    2.0    3.0
```

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

```
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
```

Problem 3. The file `ohio_1999.csv` contains data on workers in Ohio in the year 1999. Use pivot tables to answer the following questions:

1. What was the highest paid race/sex combination?
2. What race/sex combination worked the least amount of hours?
3. What race/sex combination worked the most hours per week per person?

Return a tuple for each question (in order of the questions) where the first entry is the numerical code corresponding to the race and the second entry is corresponding to the sex.

Some useful keys in understand the data are as follows:

1. In column `Sex`, {1: `male`, 2: `female`}.
2. In column `Race`, {1: `White`, 2: `African-American`, 3: `Native American/Eskimo`, 4: `Asian`}.

Discretizing Continuous Data

In the Titanic data, we 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 add `"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. Notice that when creating the pivot table, the index uses the categorical `age` instead of the column name `"Age"`.

```
# pd.cut() maps continuous entries to discrete intervals.
>>> pd.cut([1, 2, 3, 4, 5, 6, 7], [0, 4, 8])
[(4, 8], (0, 4], (0, 4], (0,4], (0, 4], (4, 8], (4, 8], (4, 8]]
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                       1.0 2.0  3.0               1.0 2.0 3.0
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 4. Use the employment data from Ohio in 1999 to answer the following questions:

1. The column `Educational Attainment` contains numbers 0-46. Any number less than 39 means the person did not get any form of degree. 39-42 refers to either a high-school or associate's degree. A number greater than 43 means the person got at least a bachelor's degree. What is the most common degree among workers?
2. Partition the `Age` column into 4 equally populated intervals. What is the most common age range among workers?
3. What age/degree combination has the smallest yearly salary on average?

Return the answer to each question (in order) as an `Interval`. For part three, the answer should be a tuple where the first entry in the `Interval` of the age and the second is the `Interval` of the degree.

An `Interval` is the object returned by `pd.cut` and `pd.qcut`. An example of getting an `Interval` from a pivot table is shown below.

```
>>> # Create pivot table used in last example with titanic dataset
>>> table = titanic.pivot_table(values="Survived",
                        index=[age], columns=[fare, "Pclass"],
                        aggfunc="count")
>>> # Get index of maximum interval
>>> table.sum(axis=1).idxmax()
Interval(0, 12, closed='right')
```

Problem 5. Examine the college dataset using pivot tables and groupby objects. Determine the answer to the following questions. If the answer is yes, save the answer as `True`. If the answer the no, save the answer as `False`. For the last question, save the answer as a string giving your explanation. Return a tuple containing your answers to the questions in order.

1. Is there a correlation between the percent of alumni that donate and the amount the school spends per student in BOTH private and public universities?
2. Partition `Grad.Rate` into intervals of 20%. Is the partition with the greatest number of schools the same for private and public universities?
3. Divide the acceptance rate into partitions of 25%. Does having a lower acceptance rate correlate with having more students from the top 10 percent of their high school class being admitted on average for BOTH private and public universities?
4. Why is the average percentage of students admitted from the top 10 percent of their high school class so high in private universities with very low acceptance rates?

12 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.*

Note

This lab will be done using Colab Notebooks. These notebooks are similar to Jupyter Notebooks but run remotely on Google's servers. Open a Google Colab notebook by going to your Google Drive account and creating a new Colaboratory file. If making a Colaboratory file is not an option, download the application Colaboratory onto your Google Drive. Once opening a new Colab Notebook, upload the file `pandas4.ipynb`. To make the data files accessible, run the following at the top of the lab:

```
>>> from google.colab import files

>>> uploaded = files.upload()
```

This will prompt you upload files for this notebook. For this lab, upload `DJIA.csv`, `paychecks.csv`, `finances.csv`, and `website_traffic.csv`.

Once the lab is complete, delete BOTH lines of code used for uploading files (the import statement and the upload statement) and download as a `.py` file to your git repository. Push the newly made `pandas4.py` file.

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.

```
>>> 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 so 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 12.1 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 12.1: 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 a `DatetimeIndex` 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 DatetimeIndex.
>>> 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 `"VALUE"` column to floats, then return the updated DataFrame.

Generating Time-based Indices

Some time series datasets come without explicit labels but 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
`start`	Starting date
`end`	End date
`periods`	Number of dates to include
`freq`	Amount of time between consecutive dates
`normalize`	Normalizes the start and end times to midnight

Table 12.2: Parameters for `pd.date_range()`.

Exactly three of the parameters `start`, `end`, `periods`, and `freq` 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 12.3 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
`"D"`	calendar daily (default)
`"B"`	business daily
`"H"`	hourly
`"T"`	minutely
`"S"`	secondly
`"MS"`	first day of the month
`"BMS"`	first weekday of the month
`"W-MON"`	every Monday
`"WOM-3FRI"`	every 3rd Friday of the month

Table 12.3: Options for the `freq` parameter to `pd.date_range()`.

```
# Create a DatetimeIndex for 5 consecutive days starting on 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')

# Create a DatetimeIndex with 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')


# Create a DatetimeIndex for 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')

# Create a DatetimeIndex for 2 hour 30 minute intervals between 4:30 PM and ↩
    2:30 AM on September 29, 2016.
>>> 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. Paychecks are given on the first and third Fridays of each month, and the employee started working on March 13, 2008.

Read in the data, using `pd.date_range()` to generate the `DatetimeIndex`. Set this as the new index of the `DataFrame` and return the `DataFrame`.

Periods

A 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 `Period` class 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 in some cases can also indicate the offset of the `Period`. The default value for `freq` is "M" for months. The `freq` parameter accepts the majority, but not all, of frequencies listed in Table 12.3.

```
# Creates a period for month of Oct, 2016.
>>> 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. After creating a `PeriodIndex`, the `freq` parameter can be changed via the `asfreq()` method.

```
# 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')

# 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')
```

The bounds of a `PeriodIndex` object can be shifted by adding or subtracting an integer. `PeriodIndex` will be shifted by $n \times$ `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. The `how` parameter can be `start` or `end` and determines if the timestamp is the beginning or the end of the period. Similarly, you can switch from timestamps to periods.

```
# 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')

>>> 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. Let `how=end`. Assume the fiscal year starts at the beginning of September and that the data begins in September 1978. Return the `DataFrame`.

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

Some datasets do not have datapoints at a fixed frequency. For example, a dataset of website traffic has datapoints that occur at irregular intervals. In situations like these, *resampling* can help provide insight on the data.

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. Then aggregation produces a new data set. The first parameter to `resample()` is an offset string from Table 12.3: `"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(days)), 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 number of visits each minute and the number of visits each hour. Return these DataFrames.

(Hint: `minute.agg(func)` returns a DataFrame).

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.

Return the index of the day with the largest gain and the day with the largest loss.
(Hint: Call `prob1()` to get the DataFrame already cleaned and with DatetimeIndex).

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 a calculation on 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.

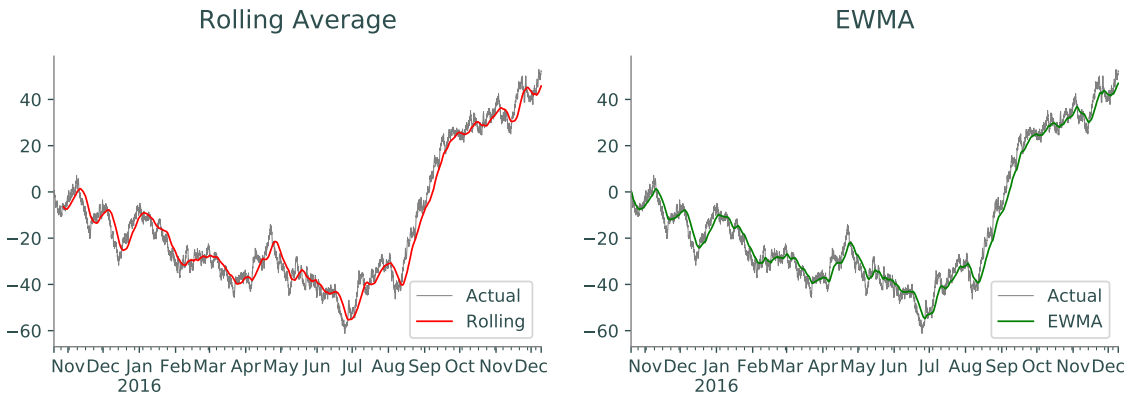
In the case of a *exponentially-weighted moving average* (EWMA), each data point is calculated as follows.

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

where z_i is the value of the EWMA at time i, $\bar{x}_i$ is the average for the i-th window, and α 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.



(a) (b)

Figure 12.1: Rolling average and EWMA.

```
ax2 = plt.subplot(122)
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.

Your plots should look like Figure 12.2. Return a list of the minimum rolling average value for each window size and a list of the minimum exponential average value for each span size.

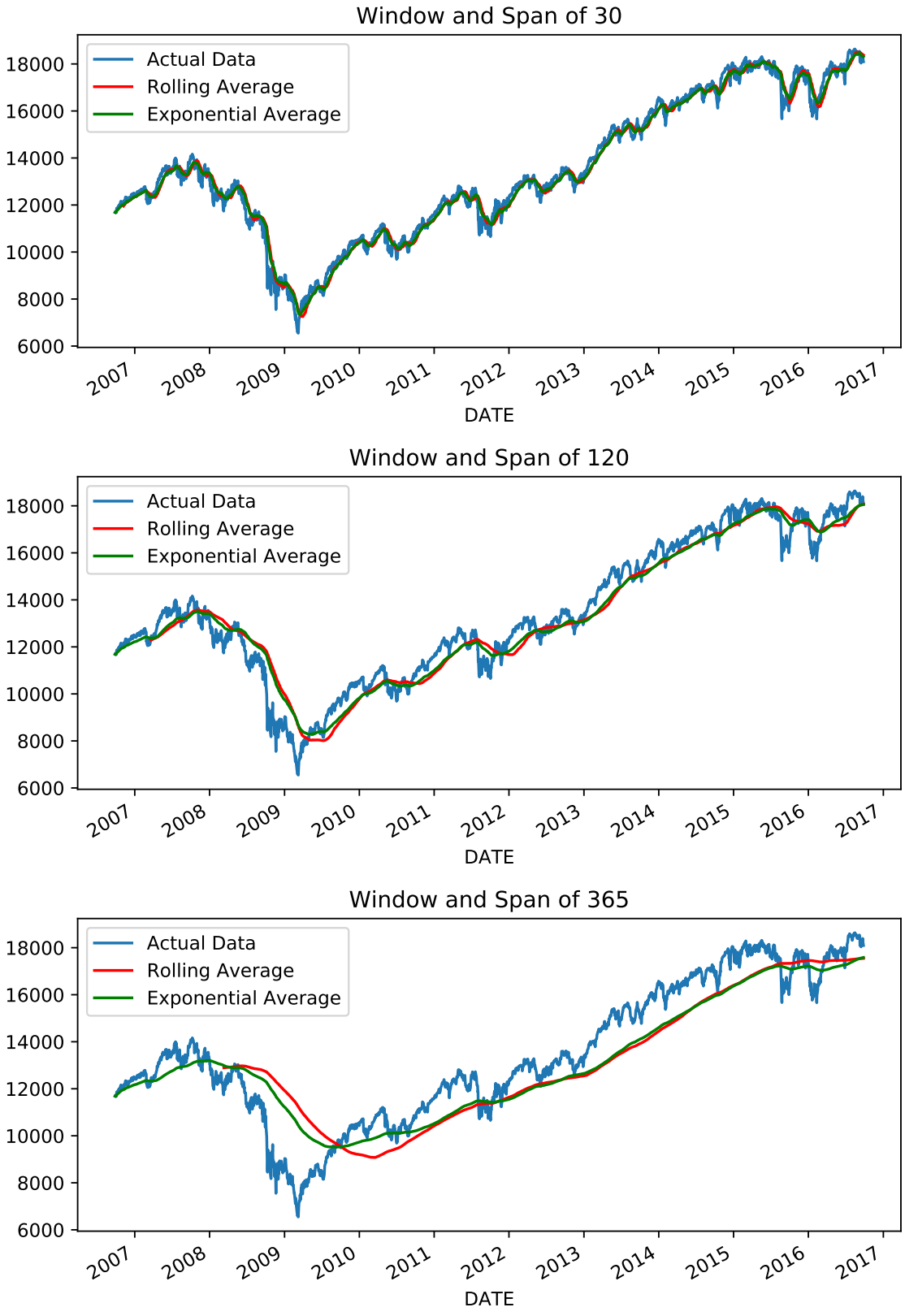


Figure 12.2: Plots for Problem 6.

13 Geopandas

Lab Objective: *Geopandas is a package designed to organize and manipulate geographic data, It combines the data manipulation tools from Pandas and the geometric capabilities of the Shapely package. In this lab, we explore the basic data structures of GeoSeries and GeoDataFrames and their functionalities.*

Installation

Geopandas is a new package designed to combine the functionalities of Pandas and Shapely, a package used for geometric manipulation. Using Geopandas with geographic data is very useful, as it allows the user to not only compare numerical data, but geometric attributes. Since Geopandas is currently under development, the installation procedure requires that all dependencies are up to date. To install Geopandas, run the following code.

```
>>> conda install geopandas
>>> conda install -c conda-forge gdal
```

A particular package needed for Geopandas is Fiona. Geopandas will not run without the correct version of this package. To check the current version of Fiona that is installed, run the following code. If the version is not at least 1.7.13, update Fiona.

```
# Check version of Fiona
>>> conda list fiona

# Update Fiona if necessary
>>> pip install fiona --upgrade
```

GeoSeries

A GeoSeries is a Pandas Series where each entry is a set of geometric objects. There are three classes of geometric objects inherited from the Shapely package:

1. Points / Multi-Points

2. Lines / Multi-Lines
3. Polygons / Multi-Polygons

A point is used to identify objects like coordinates, where there is one small instance of the object. A line could be used to describe a road, which is a collection of points. A polygon could be used to identify regions, such as a country.

Since each object in the GeoSeries is also a Shapely object, the GeoSeries inherits many methods and attributes of Shapely objects. Some of the key attributes and methods are listed in Table 13.1. These attributes and methods can be used to calculate distances, find the sizes of countries, and determine whether coordinates are within country's boundaries. The example below uses the method `bounds` to find the maximum and minimum coordinates of Egypt in a built-in GeoDataFrame.

Table 13.1: Attributes and Methods for GeoSeries

Method	Description
`distance(other)`	returns minimum distance from GeoSeries to `other`
`area`	returns shape area
`contains(other)`	returns `True` if shape is contained in `other`
`intersects(other)`	returns `True` if shape intersects `other`

```
>>> world = geopandas.read_file(geopandas.datasets.get_path('↩
    naturalearth_lowres'))
>>> # Get GeoSeries for Egypt
>>> egypt = world[world['name']=='Egypt']
>>>
>>> # Find bounds of Egypt
>>> egypt.bounds
         minx    miny         maxx      maxy
47   24.70007    22.0     36.86623  31.58568
```

Creating GeoDataFrames

The main structure used in GeoPandas is a GeoDataFrame, which is similar to a Pandas DataFrame. A GeoDataFrame has one special column called `geometry`. This GeoSeries column can be accessed through the `.geometry` attribute and is the column that is used when a spatial method, like `distance()`, is used on the GeoDataFrame.

To make a GeoDataFrame, first create a Pandas DataFrame. At least one of the columns in the DataFrame should contain geometric information. Convert a column containing geometric information to a GeoSeries using the `apply` method. At this point, the Pandas DataFrame can be cast as a GeoDataFrame. When creating a GeoDataFrame, if more than one column has geometric data, assign which column will be the `geometry` using the `set_geometry()` method.

```
import pandas as pd
import geopandas
from shapely.geometry import Point
```

```
# Create a Pandas DataFrame
df = pd.DataFrame({'City': ['Seoul', 'Lima', 'Johannesburg'],
                           'Country': ['South Korea', 'Peru', 'South Africa'],
                           'Latitude': [37.57, -12.05, -26.20],
                           'Longitude': [126.98, -77.04, 28.04]})

# Create geometry column
df['Coordinates'] = list(zip(df.Longitude, df.Latitude))

# Make geometry column Shapely objects
df['Coordinates'] = df['Coordinates'].apply(Point)

# Cast as GeoDataFrame
gdf = geopandas.GeoDataFrame(df, geometry='Coordinates')
```

Note

Longitude is the angular measurement starting at the Prime Meridian, 0°, and going to 180° to the east and −180° to the west. It is further divided into minutes and seconds. Latitude is the angle between the equatorial plane and the normal line at a given point; a point along the Equator has latitude 0, the North Pole has latitude +90° or 90°N, and the South Pole has latitude −90° or 90°S.

In order to find the distance between two points in latitude and longitude using the distance function, it is necessary to use the latitude and longitude to convert the points from spherical to cartesian coordinates. Recall that

$$\begin{aligned} x &= \rho \sin(\theta)\cos(\phi) \\ y &= \rho \sin(\theta)\cos(\phi) \\ z &= \rho \cos(\phi). \end{aligned}$$

This means that the code above must be modified such that the geometry column is a zipped list of cartesian coordinates rather than latitudes and longitudes.

Problem 1. Read in the file `airports.csv` as a Pandas DataFrame. Convert the DataFrame into a GeoDataFrame. (Set the geometry column as Point objects).

Find the distance between the following airports:

1. Halifax / CFB Shearwater Heliport (Halifax, Canada) to Murtala Muhammed International Airport (Lagos, Nigeria)
2. Don Mueang International Airport (Bangkok, Thailand) to Beijing Capital International Airport (Beijing, China)

3. Salt Lake City International Airport (Salt Lake City, USA) to Auckland International Airport (Auckland, New Zealand)

GeoDataFrames

As previously mentioned, GeoDataFrames contain many of the functionalities of Pandas DataFrames. For example, to create a new column, define a new column name in the GeoDataFrame with the needed information for each GeoSeries.

```
# Create column in world GeoDataFrame for gdp_per_capita
world['gdp_per_cap'] = world.gdp_md_est / world.pop_est
```

While many Pandas functionalities are useful with GeoDataFrames, they can also be parsed by geometric manipulations. For example, a useful way to index GeoDataFrames is with the `cx` indexer. This splits the GeoDataFrame by the coordinates of each geometric object. It is used by calling the method `cx` on a GeoDataFrame, followed by a slicing argument, where the first element refers to the longitude and the second refers to latitude.

```
# Create a GeoDataFrame containing the northern hemisphere
north = world.cx[: , 0:]

# Create a GeoDataFrame containing the southeastern hemisphere
south_east = world.cx[0: , :0]
```

GeoSeries in a GeoDataFrame can also be dissolved, or merged, together into one GeoSeries based on their geometry data. For example, all countries on one continent could be merged to create a GeoSeries containing the information of that continent. The method designed for this is called `dissolve`. It receives two parameters, `by` and `aggfunc`. `by` indicates which column to dissolve according and `aggfunc` tells how to combine the information in all other columns. The default `aggfunc` is `first`, which returns the first application entry.

```
world = world[['continent', 'geometry', 'gdp_per_cap']]

# Dissolve world GeoDataFrame by continent
continent = world.dissolve(by = 'continent', aggfunc='sum')
```

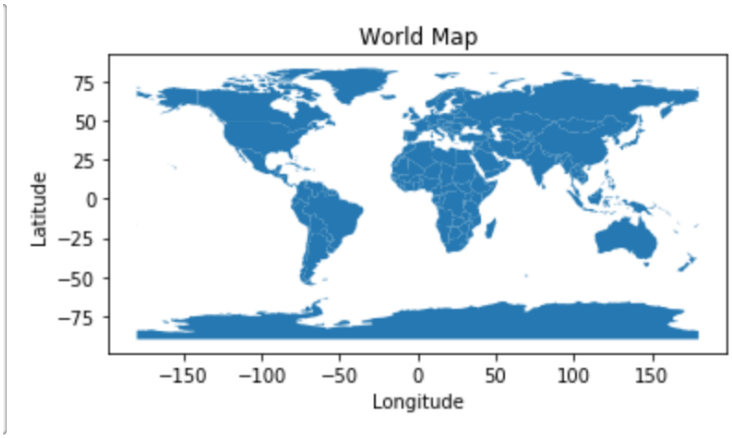
Problem 2. Read in the built-in GeoDataFrame `naturalearth_lowres`. Create a GeoDataFrame that only contains information about the southern hemisphere. Use this data to find the countries with the smallest and largest area in the southern hemisphere. Dissolve this GeoDataFrame to find the continent with the largest and smallest area in the southern hemisphere.

Plotting with GeoPandas

GeoDataFrames can be easily plotted with GeoPlot. GeoPlot plots the information from a GeoDataFrame based on their geometry column and displays the data as geometry objects. To use GeoPlots, import `geoplot`.

```
>>> import geoplot

>>> # Plot world GeoDataFrame
>>> world.plot()
```



With GeoPlot, multiple GeoDataFrames can be plotted at once. For example, if the information in a GeoDataFrame contains the coordinates of capitals of countries, it can be plotted on top of a GeoDataFrame containing the polygons of country boundaries to show a world map with capitals for each country. This is done by by setting one GeoDataFrame as the base of the GeoPlot.

```
>>> # Set world map as base
>>> base = world.plot(color='white', edgecolor='black')

>>> # Plot airports on world map
>>> airport.plot(ax=base, marker='o', color='green', markersize=5)
```

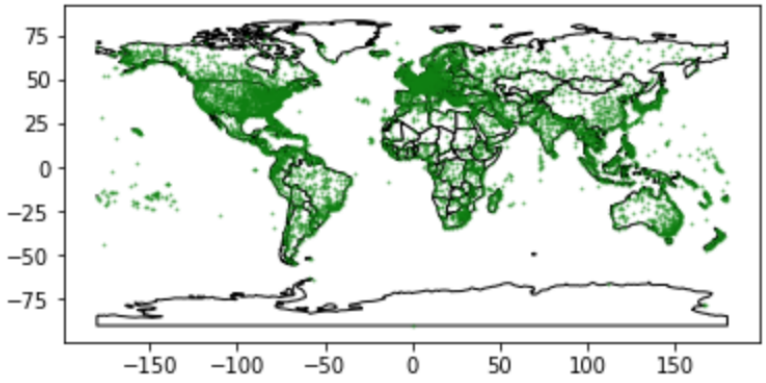


Figure 13.1: Airport-map

When plotting, GeoPlot refers to the CRS (coordinate reference system) of a GeoDataFrame. This reference system informs how coordinates should be spaced on a plot. GeoPandas accepts many different CRSs, and references to them can be found at `www.spatialreference.org`. Two of the most commonly used CRS are WGS84 and WGS85. The crs of WGS84 is ESPG:4326, and it does the standard latitude-longitude projection used by GPS. WGS85, also known as Mercator and EPSG:3395 is the standard navigational projection.

When creating a new GeoDataFrame, it is important to set the crs attribute of the GeoDataFrame. This allows the plot to be done correctly. GeoDataFrames being layered should also have the same CRS.

```
>>> import geoplot.crs as gcrs

>>> # Check crs of world GeoDataFrame
>>> world.crs
{'init': 'epsg:4326'}

# Change CRS of world to Mercator
>>> world.to_crs({'init': 'epsg:3395'}
>>> world.crs
{'init': 'epsg:3395']
```

GeoDataFrames can also be plotted using GeoPlot by the values in the the other attributes of the GeoSeries. This is done with a Choropleth map. The map plots the color of each geometry object according to the value of the column selected. This is done by passing in the parameter `column` into the plot method.

```
>>> # Plot world based on gdp
>>> world.plot(column='gdp_md_est', cmap='OrRd')
```

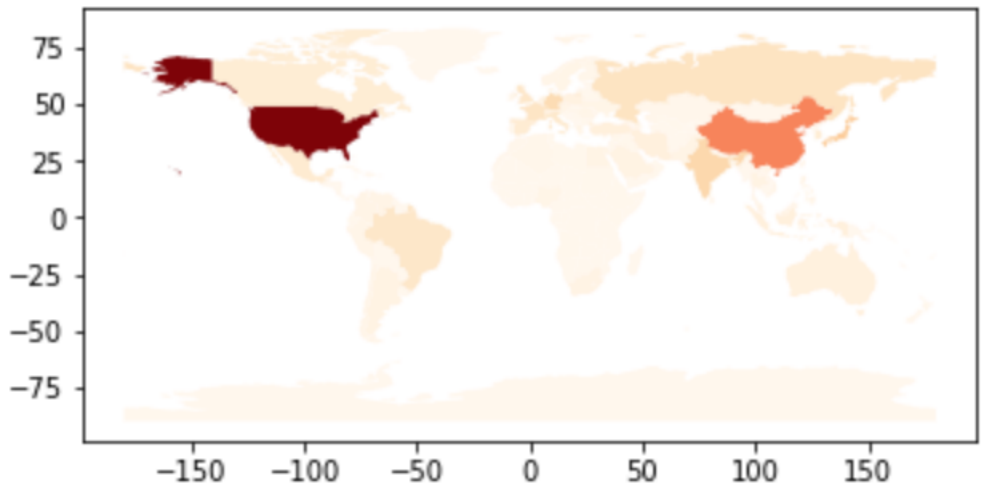


Figure 13.2: World Map Based on GDP

Problem 3. Using the built-in GeoDataFrame `naturalearth_lowres` and GeoPlots, create population density plots for Asia and South America. Use Mercator maps.

Merging GeoDataFrames

As Pandas DataFrames can be merged, GeoDataFrames can be joined on either attributes or spatial joins. An attribute join is similar to that of a merge in Pandas. It combines two GeoDataFrames on a column (not the geometry column) and then combines the rest of the data into one GeoDataFrame.

```
>>> world = geopandas.read_file(geopandas.datasets.get_path('↩
    naturalearth_lowres'))
>>> cities = geopandas.read_file(geopandas.datasets.get_path('↩
    naturalearth_cities'))

>>> # Create subsets of the world and cities GeoDataFrames
>>> world = world[['continent', 'name', 'iso_a3']]
>>> cities = cities[['name', 'iso_a3']]

>>> # Attribute join the GeoDataFrames on their iso_a3 code
>>> # (The iso_a3 code for Afghanistan is AFG)

>>> countries = world.merge(cities, on='iso_a3')
```

A spatial join merges two GeoDataFrames based on their geometry data. The function used for this is `sjoin`. `sjoin` accepts the two GeoDataFrames desired to be merged and then direction on how to do the merge. It is imperative that two GeoDataFrames being joined spatially have the same CRS. In the example below, we merge with an inner join with the option `intersects`. The inner join tells us that we will use both geometry columns and retain only the left geometry column. Intersects tells the GeoDataFrames to merge on GeoSeries that intersect each other.

```
>>> # Combine countries and cities on their geographic location
>>> # This will combine countries with their capital

countries = geopandas.sjoin(world, cities, how='inner', op='intersects')
```

Problem 4. Merge the airports GeoDataFrame and the world GeoDataFrame on their spatial data. Use this new GeoDataFrame to find the airport in the country with the smallest population estimate.

14 MongoDB

Lab Objective: *Relational databases, including those managed with SQL or pandas, require data to be organized into tables. However, many data sets have an inherently dynamic structure that cannot be efficiently represented as tables. MongoDB is a non-relational database management system that is well-suited to large, fast-changing datasets. In this lab we introduce the Python interface to MongoDB, including common commands and practices.*

Database Initialization

Suppose the manager of a general store has all sorts of inventory: food, clothing, tools, toys, etc. There are some common attributes shared by all items: name, price, and producer. However, other attributes are unique to certain items: sale price, number of wheels, or whether or not the product is gluten-free. A relational database housing this data would be full of mostly-blank rows, which is extremely inefficient. In addition, adding new items to the inventory requires adding new columns, causing the size of the database to rapidly increase. To efficiently store the data, the whole database would have to be restructured and rebuilt often.

To avoid this problem, NoSQL databases like MongoDB avoid using relational tables. Instead, each item is a JSON-like object, and thus can contain whatever attributes are relevant to the specific item, without including any meaningless attribute columns.

> **Note**
>
> MongoDB is a database management system (DBMS) that runs on a server, which should be running in its own dedicated terminal. Refer to the Additional Material section for installation instructions.

The Python interface to MongoDB is called `pymongo`. After installing `pymongo` **and** with the MongoDB server running, use the following code to connect to the server.

```
>>> from pymongo import MongoClient
# Create an instance of a client connected to a database running
# at the default host IP and port of the MongoDB service on your machine.
>>> client = MongoClient()
```

Creating Collections and Documents

A MongoDB database stores *collections*, and a collection stores *documents*. The syntax for creating databases and collections is a little unorthodox, as it is done through attributes instead of methods.

```
# Create a new database.
>>> db = client.db1

# Create a new collection in the db database.
>>> col = db.collection1
```

Documents in MongoDB are represented as JSON-like objects, and therefore do not adhere to a set schema. Each document can have its own *fields*, which are completely independent of the fields in other documents.

```
# Insert one document with fields 'name' and 'age' into the collection.
>>> col.insert_one({'name': 'Jack', 'age': 23})

# Insert another document. Notice that the value of a field can be a string,
# integer, truth value, or even an array.
>>> col.insert_one({'name': 'Jack', 'age': 22, 'student': True,
...                 'classes': ['Math', 'Geography', 'English']})

# Insert many documents simultaneously into the collection.
>>> col.insert_many([
...     {'name': 'Jill', 'age': 24, 'student': False},
...     {'name': 'John', 'nickname': 'Johnny Boy', 'soldier': True},
...     {'name': 'Jeremy', 'student': True, 'occupation': 'waiter'}  ])
```

Note

Once information has been added to the database it will remain there, even if the python environment you are working with is shut down. It can be reaccessed anytime using the same commands as before.

```
>>> client = MongoClient()
>>> db = client.db1
>>> col = db.collection1
```

To delete a collection, use the database's `drop_collection()` method. To delete a database, use the client's `drop_database()` method.

Problem 1. The file `trump.json` contains posts from `http://www.twitter.com` (tweets) over the course of an hour that have the key word "trump".[a] Each line in the file is a single JSON message that can be loaded with `json.loads()`.

Create a MongoDB database and initialize a collection in the database. Use the collection's `delete_many()` method with an empty set as input to clear existing contents of the collection, then fill the collection one line at a time with the data from `trump.json`. Check that your collection has 95,643 entries with its `count()` method.

[a] See the Additional Materials section for an example of using the Twitter API.

Querying a Collection

MongoDB uses a *query by example* pattern for querying. This means that to query a database, an example must be provided for the database to use in matching other documents.

```
# Find all the documents that have a 'name' field containing the value 'Jack'.
>>> data = col.find({'name': 'Jack'})

# Find the FIRST document with a 'name' field containing the value 'Jack'.
>>> data = col.find_one({'name': 'Jack'})
```

The `find_one()` method returns the first matching document as a dictionary. The `find()` query may find any number of objects, so it will return a `Cursor`, a Python object that is used to iterate over the query results. There are many useful functions that can be called on a `Cursor`, for more information see `http://api.mongodb.com/python/current/api/pymongo/cursor.html`.

```
# Search for documents containing True in the 'student' field.
>>> students = col.find({'student': True})
>>> students.count()                # There are 2 matching documents.
2

# List the first student's data.
# Notice that each document is automatically assigned an ID number as '_id'.
>>> students[0]
{'_id': ObjectId('59260028617410748cc7b8c7'),
 'age': 22,
 'classes': ['Math', 'Geography', 'English'],
 'name': 'Jack',
 'student': True}

# Get the age of the first student.
>>> students[0]['age']
22

# List the data for every student.
>>> list(students)
[{'_id': ObjectId('59260028617410748cc7b8c7'),
```

```
  'age': 22,
  'classes': ['Math', 'Geography', 'English'],
  'name': 'Jack',
  'student': True},
 {'_id': ObjectId('59260028617410748cc7b8ca'),
  'name': 'Jeremy',
  'occupation': 'waiter',
  'student': True}]
```

The Logical operators listed in the following table can be used to do more complex queries.

Operator	Description
$lt, $gt	<, >
$lte,$gte	<=, >=
$eq, $ne	==, !=
$in, $nin	in, not in
$or, $and, $not	or, and, not
$exists	Match documents with a specific field
$type	Match documents with values of a specific type
$all	Match arrays that contain all queried elements
$size	Match arrays with a specified number of elements
$regex	Search documents with a regular expression

Table 14.1: MongoDB Query Operators

```
# Query for everyone that is either above the age of 23 or a soldier.
>>> results = col.find({'$or':[{'age':{'$gt': 23}},{'soldier': True}]})

# Query for everyone that is a student (those that have a 'student' attribute
# and haven't been expelled).
>>> results = col.find({'student': {'$not': {'$in': [False, 'Expelled']}}})

# Query for everyone that has a student attribute.
>>> results = col.find({'student': {'$exists': True}})

# Query for people whose name contains a the letter 'e'.
>>> import re
>>> results = col.find({'name': {'$regex': re.compile('e')}})
```

It is likely that a database will hold more complex JSON entries then these, with many nested attributes and arrays. For example, an entry in a database for a school might look like this.

```
{'name': 'Jason', 'age': 16,
 'student': {'year':'senior', 'grades': ['A','C','A','B'],'flunking': False},
 'jobs':['waiter', 'custodian']}
```

To query the nested attributes and arrays use a dot, as in the following examples.

```
# Query for student that are seniors
>>> results = col.find({'student.year': 'senior'})

# Query for students that have an A in their first class.
>>> results = col.find({'student.grades.0': 'A'})
```

The Twitter JSON files are large and complex. To see what they look like, either look at the JSON file used to populate the `collection` or print any tweet from the database. The following website also contains useful information about the fields in the JSON file `https://dev.twitter.com/overview/api/tweets`.

The `distinct` function is also useful in seeing what the possible values are for a given field.

```
# Find all the values in the names field.
>>> col.distinct("name")
['Jack', 'Jill', 'John', 'Jeremy']
```

Problem 2. Query the Twitter collection from Problem 1 for the following information.

- How many tweets include the word Russia? Use `re.IGNORECASE`.
- How many tweets came from one of the main continental US time zones? These are listed as `"Central Time (US & Canada)"`, `"Pacific Time (US & Canada)"`, `"Eastern Time (US & Canada)"`, and `"Mountain Time (US & Canada)"`.
- How often did each language occur? Construct a dictionary with each language and it's frequency count.
  (Hint: use `distinct()` to get the language options.)

Deleting and Sorting Documents

Items can be deleted from a database using the same syntax that is used to find them. Use `delete_one` to delete just the first item that matches your search, or `delete_many` to delete all items that match your search.

```
# Delete the first person from the database whose name is Jack.
>>> col.delete_one({'name':'Jack'})

# Delete everyone from the database whose name is Jack.
>>> col.delete_many({'name':'Jack'})

# Clear the entire collection.
>>> col.delete_many({})
```

Another useful function is the `sort` function, which can sort the data by some attribute. It takes in the attribute by which the data will be sorted, and then the direction (1 for ascending and -1 for descending). Ascending is the default. The following code is an example of sorting.

```
# Sort the students by name in alphabetic order.
>>> results = col.find().sort('name', 1)
>>> for person in results:
...     print(person['name'])
...
Jack
Jack
Jeremy
Jill
John

# Sort the students oldest to youngest, ignoring those whose age is not listed.
>>> results = col.find({'age': {'$exists': True}}).sort('age', -1)
>>> for person in results:
...    print(person['name'])
...
Jill
Jack
Jack
```

Problem 3. Query the Twitter collection from Problem 1 for the following information.

- What are the usernames of the 5 most popular (defined as having the most followers) tweeters? Don't include repeats.
- Of the tweets containing at least 5 hashtags, sort the tweets by how early the 5th hashtag appears in the text. What is the earliest spot (character count) it appears?
- What are the coordinates of the tweet that came from the northernmost location? Use the latitude and longitude point in `"coordinates"`.

Updating Documents

Another useful attribute of MongoDB is that data in the database can be updated. It is possible to change values in existing fields, rename fields, delete fields, or create new fields with new values. This gives much more flexibility than a relational database, in which the structure of the databse must stay the same. To update a database, use either `update_one` or `update_many`, depending on whether one or more documents should be changed (the same as with `delete`). Both of these take two parameters; a find query, which finds documents to change, and the update parameters, telling these things what to update. The syntax is `update_many({find query},{update parameters})`.

The update parameters must contain update operators. Each update operator is followed by the field it is changing and the value to change it. The syntax is the same as with query operators. The operators are shown in the table below.

Operator	Description
`$inc` , `$mul`	`+=`, `*=`
`$min`, `$max`	`min()`, `max()`
`$rename`	Rename a specified field to the given new name
`$set`	Assign a value to a specified field (creating the field if necessary)
`$unset`	Remove a specified field
`$currentDate`	Set the value of the field to the current date. With `"$type": "date"`, use a `datetime` format; with `"$type": "timestamp:`, use a timestamp.

Table 14.2: MongoDB Update Operators

```
# Update the first person from the database whose name is Jack to include a
# new field 'lastModified' containing the current date.
>>> col.update_one({'name':'Jack'},
...                {'$currentDate': {'lastModified': {'$type': 'date'}}})

# Increment everyones age by 1, if they already have an age field.
>>> col.update_many({'age': {'$exists': True}}, {'$inc': {'age': 1}})

# Give the first John a new field 'best_friend' that is set to True.
>>> col.update_one({'name':'John'}, {'$set': {'best_friend': True}})
```

Problem 4. Clean the twitter collection in the following ways.

- Get rid of the `"retweeted_status"` field in each tweet.
- Update every tweet from someone with at least 1000 followers to include a `popular` field whose value is `True`. Report the number of popular tweets.
- (OPTIONAL) The geographical coordinates used before in `coordinates.coordinates` are turned off for most tweets. But many more have a bounding box around the coordinates in the `place` field. Update every tweet without coordinates that contains a bounding box so that the coordinates contains the average value of the points that form the bounding box. Make the structure of `coordinates` the same as the others, so it contains `coordinates` with a longitude, latitude array and a `type`, the value of which should be 'Point'.
  (Hint: Iterate through each tweet in with a bounding box but no coordinates. Then for each tweet, grab it's id and the bounding box coordinates. Find the average, and then update the tweet. To update it search for it's id and then give the needed update parameters. First unset coordinates, and then set coordinates.coordinates and coordinates.type to the needed values.)

Additional Material

Installation of MongoDB

MongoDB runs as an isolated program with a path directed to its database storage. To run a practice MongoDB server on your machine, complete the following steps:

Create Database Directory

To begin, navigate to an appropriate directory on your machine and create a folder called `data`. Within that folder, create another folder called `db`. Make sure that you have read, write, and execute permissions for both folders.

Retrieve Shell Files

To run a server on your machine, you will need the proper executable files from MongoDB. The following instructions are individualized by operating system. For all of them, download your binary files from `https://www.mongodb.com/download-center?jmp=nav#community`.

1. For Linux/Mac:

   Extract the necessary files from the downloaded package. In the terminal, navigate into the `bin` directory of the extracted folder. You may then start a Mongo server by running in a terminal: `./mongod --dbpath /pathtoyourdatafolder`.

2. For Windows:

   Go into your Downloads folder and run the Mongo `.msi` file. Follow the installation instructions. You may install the program at any location on your machine, but do not forget where you have installed it. You may then start a Mongo server by running in command prompt: C:\locationofmongoprogram\mongod.exe –dbpath C:\pathtodatafolder\data\db.

MongoDB servers are set by default to run at address:port `127.0.0.1:27107` on your machine.

You can also run Mongo commands through a mongo terminal shell. More information on this can be found at `https://docs.mongodb.com/getting-started/shell/introduction/`.

Twitter API

Pulling information from the Twitter API is simple. First you must get a Twitter account and register your app with them on `apps.twitter.com`. This will enable you to have a consumer key, consumer secret, access token, and access secret, all required by the Twitter API.

You will also need to install `tweepy`, an open source library that allows python to easily work with the Twitter API. This can be installed with pip by running from the command line

```
$pip install tweepy
```

The data for this lab was then pulled using the following code on May 26, 2017.

```
import tweepy
from tweepy import OAuthHandler
from tweepy import Stream
```

```
from tweepy.streaming import StreamListener
from pymongo import MongoClient
import json

#Set up the databse
client = MongoClient()
mydb = client.db1
twitter = mydb.collection1

f = open('trump.txt','w') #If you want to write to a file

consumer_key = #Your Consumer Key
consumer_secret = #Your Consumer Secret
access_token = #Your Access Token
access_secret = #Your Access Secret

my_auth = OAuthHandler(consumer_key, consumer_secret)
my_auth.set_access_token(access_token, access_secret)

class StreamListener(tweepy.StreamListener):
    def on_status(self, status):
        print(status.text)

    def on_data(self, data):
        try:
            twitter.insert_one(json.loads(data)) #Puts the data into your ↩
                MongoDB
            f.write(str(data)) #Writes the data to an output file
            return True
        except BaseException as e:
            print(str(e))
            print("Error")
        return True

    def on_error(self, status):
        print(status)
        if status_code == 420: #This means twitter has blocked us temporarily, ↩
            so we want to stop or they will get mad. Wait 30 minutes or so and ↩
            try again. Running this code often in a short period of time will ↩
            cause twitter to block you. But you can stream tweets for as long ↩
            as you want without any problems.
            return False
        else:
            return True

stream_listener = StreamListener()
stream = tweepy.Stream(auth=my_auth, listener=stream_listener)
stream.filter(track=["trump"]) #This pulls all tweets that include the keyword ↩
    "trump". Any number of keywords can be searched for.
```

15 Introduction to Parallel Computing

Lab Objective: *Many modern problems involve so many computations that running them on a single processor is impractical or even impossible. There has been a consistent push in the past few decades to solve such problems with parallel computing, meaning computations are distributed to multiple processors. In this lab, we explore the basic principles of parallel computing by introducing the cluster setup, standard parallel commands, and code designs that fully utilize available resources.*

Parallel Architectures

A *serial* program is executed one line at a time in a single process. Since modern computers have multiple processor cores, serial programs only use a fraction of the computer's available resources. This can be beneficial for smooth multitasking on a personal computer because programs can run uninterrupted on their own core. However, to reduce the runtime of large computations, it is beneficial to devote all of a computer's resources (or the resources of many computers) to a single program. In theory, this parallelization strategy can allow programs to run N times faster where N is the number of processors or processor cores that are accessible. Communication and coordination overhead prevents the improvement from being quite that good, but the difference is still substantial.

A *supercomputer* or *computer cluster* is essentially a group of regular computers that share their processors and memory. There are several common architectures that combine computing resources for parallel processing, and each architecture has a different protocol for sharing memory and processors between *computing nodes*, the different simultaneous processing areas. Each architecture offers unique advantages and disadvantages, but the general commands used with each are very similar.

The iPyParallel Architecture

In most circumstances, processors communicate and coordinate with a message-passing system such as the standard *Message Passing Interface* (MPI). Many basic commands used for parallel programming with MPI are implemented by Python's `ipyparallel` package. There are three main parts of the iPyParallel architecture.

- *Client*: The main human-written program that is being run.

- *Controller*: Receives directions from the client and distributes instructions and data to the computing nodes. Consists of a *hub* to manage communications and *schedulers* to assign processes to the engines.

- *Engines*: The individual computing nodes. Each engine is like a separate Python process, each with its own namespace, and computing resources.

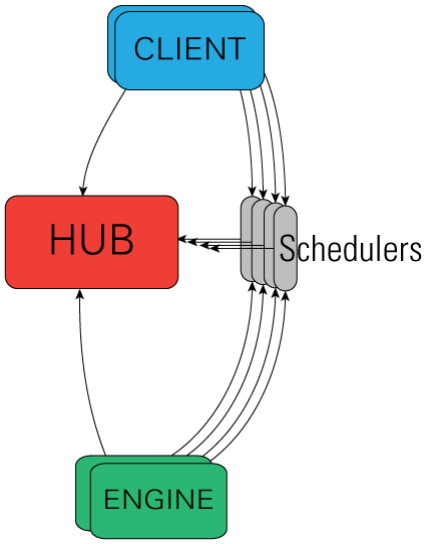


Figure 15.1: An outline of the iPyParallel architecture.

Setting up an iPyParallel Cluster

Anaconda includes iPyParallel, so it can be installed with `conda install ipyparallel`. Establishing a cluster on multiple machines requires a bit of extra setup, which is detailed in the Additional Material section. For now, we demonstrate how to use a single machine with multiple processor cores as a cluster. The following commands initialize parts or all of a cluster in a terminal window.

Command	Description
`ipcontroller start`	Initialize a controller process.
`ipengine start`	Initialize an engine process.
`ipcluster start`	Initialize a controller process and several engines simultaneously.

Each of these processes can be stoped with a keyboard interrupt (`Ctrl+C`). By default, the controller uses JSON files in `UserDirectory/.ipython/profile-default/security/` to determine its settings. Once a controller is running, it acts like a server listening for client connections from engine processes. Engines connect by default to a controller with the settings defined in the aforementioned JSON files. There is no limit to the number of engines that can be started in their own terminal windows and connected to the controller, but it is recommended to only use as many engines areas there are cores to maximize efficiency. Once started, each engine has its own ID number on the controller that is used for communication.

Achtung!

The directory that the controller and engines are started from matters. To facilitate connections, navigate to the same folder as your source code before using `ipcontroller`, `ipengine`, or `ipcluster`. Otherwise, the engines may not connect to the controller or may not be able to find auxiliary code as directed by the client.

Starting a controller and engines in individual terminal windows with `ipcontroller` and `ipengine` is a little inconvenient, but having separate terminal windows for the engines allows the user to see individual errors in detail. It is also actually more convenient when starting a cluster of multiple computers. For now, we use `ipcluster` to get the entire cluster started quickly.

```
$ ipcluster start           # Assign an engine to each processor core.
$ ipcluster start --n 4     # Or, start a cluster with 4 engines.
```

Note

Jupyter notebooks also have a **Clusters** tab in which clusters can be initialized using an interactive GUI. To enable the tab, run the following command. This operation may require root permissions.

```
$ ipcluster nbextension enable
```

The iPyParallel Interface

Once a controller and its engines have been started and are connected, a cluster has successfully been established. The controller will then be able to distribute messages to each of the engines, which will compute with their own processor and memory space and return their results to the controller. The client uses the `ipyparallel` module to send instructions to the controller via a `Client` object.

```
>>> from ipyparallel import Client

>>> client = Client()       # Only works if a cluster is running.
>>> client.ids
[0, 1, 2, 3]                # Indicates that there are four engines running.
```

Once the client object has been created, it can be used to create one of two classes: a `DirectView` or a `LoadBalancedView`. These views allow for messages to be sent to collections of engines simultaneously. A `DirectView` allows for total control of task distribution while a `LoadBalancedView` automatically tries to spread out the tasks equally on all engines. The remainder of the lab will be focused on the `DirectView` class.

```
>>> dview = client[:]   # Group all engines into a DirectView.
>>> dview2 = client[:2] # Group engines 0,1, and 2 into a DirectView.
```

```
>>> dview2.targets      # See which engines are connected.
[0, 1, 2]
```

Since each engine has its own namespace, modules must be imported in every engine. There is more than one way to do this, but the easiest way is to use the `DirectView` object's `execute()` method, which accepts a string of code and executes it in each engine.

```
# Import NumPy in each engine.
>>> dview.execute("import numpy as np")
```

Problem 1. Write a function that initializes a `Client` object, creates a `DirectView` with all available engines, and imports `scipy.sparse` as `sparse` on all engines.

Managing Engine Namespaces

Push and Pull

The `push()` and `pull()` methods of a `DirectView` object manage variable values in the engines. Use `push()` to set variable values and `pull()` to get variables. Each method has an easy shortcut via indexing.

```
>>> dview.block = True      # IMPORTANT!! Blocking will be explained later.

# Initialize the variables 'a' and 'b' on each engine.
>>> dview.push({'a':10, 'b':5})         # OR dview['a'] = 10; dview['b'] = 5
[None, None, None, None]

# Check the value of 'a' on each engine.
>>> dview.pull('a')                     # OR dview['a']
[10, 10, 10, 10]

# Put a new variable 'c' only on engines 0 and 2.
>>> dview.push({'c':12}, targets=[0, 2])
[None, None]
```

Problem 2. Write a function `variables(dx)` that accepts a dictionary of variables. Create a `Client` object and a `DirectView` and distribute the variables. Pull the variables back and make sure they haven't changed.

Scatter and Gather

Parallelization almost always involves splitting up collections and sending different pieces to each engine for processing. The process is called *scattering* and is usually used for dividing up arrays or lists. The inverse process of pasting a collection back together is called *gathering*. This method of distributing and collecting a dataset is the foundation of the prominent MapReduce algorithm.

```
>>> import numpy as np

# Send parts of an array of 8 elements to each of the 4 engines.
>>> x = np.arange(1, 9)
>>> dview.scatter("nums", x)
>>> dview["nums"]
[array([1, 2]), array([3, 4]), array([5, 6]), array([7, 8])]

# Scatter the array to only the first two engines.
>>> dview.scatter("nums_big", x, targets=[0,1])
>>> dview.pull("nums_big", targets=[0,1])
[array([1, 2, 3, 4]), array([5, 6, 7, 8])]

# Gather the array again.
>>> dview.gather("nums")
array([1, 2, 3, 4, 5, 6, 7, 8])

>>> dview.gather("nums_big", targets=[0,1])
array([1, 2, 3, 4, 5, 6, 7, 8])
```

Executing Code on Engines

Execute

The `execute()` method is the simplest way to run commands on parallel engines. It accepts a string of code (with exact syntax) to be executed. Though simple, this method works well for small tasks.

```
# 'nums' is the scattered version of np.arange(1, 9).
>>> dview.execute("c = np.sum(nums)")   # Sum each scattered component.
<AsyncResult: execute:finished>
>>> dview['c']
[3, 7, 11, 15]
```

Apply

The `apply()` method accepts a function and arguments to plug into it, and distributes them to the engines. Unlike `execute()`, `apply()` returns the output from the engines directly.

```
>>> dview.apply(lambda x: x**2, 3)
[9, 9, 9, 9]
>>> dview.apply(lambda x,y: 2*x + 3*y, 5, 2)
[16, 16, 16, 16]
```

Note that the engines can access their local variables in any of the execution methods.

Problem 3. Write a function that accepts an integer n. Instruct each engine to make n draws from the standard normal distribution, then hand back the minimum, maximum, and mean draw to the client. Print the results. If you have four engines running, your output should resemble the following:

```
means = [0.0031776784, -0.0058112042, 0.0012574772, -0.0059655951]
maxs = [4.0388107, 4.3664958, 4.2060184, 4.3391623]
mins = [-4.1508589, -4.3848019, -4.1313324, -4.2826519]
```

Problem 4. Use your function from Problem 3 to compare serial and parallel execution times. For $n = 1000000, 5000000, 10000000, 15000000$,

1. Time how long it takes to run your function.
2. Time how long it takes to do the same process (make n draws and calculate and record the statistics) in a for loop with N iterations, where N is the number of engines running.

Plot the execution times against n. You should notice an increase in efficiency in the parallel version as the problem size increases.

Map

The built-in `map()` function applies a function to each element of an iterable. The iPyParallel equivalent, the `map()` method of the `DirectView` class, combines `apply()` with `scatter()` and `gather()`. Simply put, it accepts a dataset, splits it between the engines, executes a function on the given elements, returns the results, and combines them into one object. This function also represents a key component in the MapReduce algorithm.

```
>>> num_list = [1, 2, 3, 4, 5, 6, 7, 8]
>>> def triple(x):                  # Map a function with a single input.
...     return 3*x
...
>>> dview.map(triple, num_list)
[3, 6, 9, 12, 15, 18, 21, 24]

>>> def add_three(x, y, z):         # Map a function with multiple inputs.
...     return x+y+z
...
>>> x_list = [1, 2, 3, 4]
>>> y_list = [2, 3, 4, 5]
>>> z_list = [3, 4, 5, 6]
>>> dview.map(add_three, x_list, y_list, z_list)
[6, 9, 12, 15]
```

Blocking vs. Non-Blocking

Parallel commands can be implemented two ways. The difference is subtle but extremely important.

- *Blocking*: The controller places commands on the specified engines' execution queues, then "blocks" execution until every engine finishes its task. The main program halts until the answer is received from the controller. This mode is usually best for problems in which each node is performing the same task.
- *Non-Blocking*: The controller places commands on the specified engines' execution queues, then immediately returns an `AsyncResult` object that can be used to check the execution status and eventually retrieve the actual result. The main program continues without waiting for responses.

The execution methods `execute()`, `apply()`, and `map()`, as well as `push()`, `pull()`, `scatter()`, and `gather()`, each have a keyword argument `block` that specifies whether or not to using blocking. If not specified, the argument defaults to the `block` attribute of the `DirectView`. Alternatively, the methods `apply_sync()` and `map_sync()` always use blocking, and `apply_async()` and `map_async()` always use non-blocking.

```
>>> f = lambda n: np.sum(np.random.random(n))

# Evaluate f(n) for n=0,1,...,999 with blocking.
>>> %time block_results = [dview.apply_sync(f, n) for n in range(1000)]
CPU times: user 9.64 s, sys: 879 ms, total: 10.5 s
Wall time: 13.9 s

# Evaluate f(n) for n=0,1,...,999 with non-blocking.
>>> %time responses = [dview.apply_async(f, n) for n in range(1000)]
CPU times: user 4.19 s, sys: 294 ms, total: 4.48 s
Wall time: 7.08 s

# The non-blocking method is faster, but we still need to get its results.
>>> block_results[1]        # This list holds actual result values.
[5.9734047365913572,
 5.1895936886345959,
 4.9088268102823909,
 4.8920224621657855]
>>> responses[10]           # This list holds AsyncResult objects.
<AsyncResult: <lambda>:finished>
>>> %time nonblock_results = [r.get() for r in responses]
CPU times: user 3.52 ms, sys: 11 mms, total: 3.53 ms
Wall time: 3.54 ms          # Getting the responses takes little time.
```

As was demonstrated above, when non-blocking is used, commands can be continuously sent to engines before they have finished their previous task. This allows them to begin their next task without waiting to send their calculated answer and receive a new command. However, this requires a design that incorporates check points to retrieve answers and enough memory to store response objects.

Table 15.1 details the methods of the `AsyncResult` object.

Class Method	Description
`wait(timeout)`	Wait until the result is available or until `timeout` seconds pass. This method always returns `None`.
`ready()`	Return whether the call has completed.
`successful()`	Return whether the call completed without raising an exception. Will raise `AssertionError` if the result is not ready.
`get(timeout)`	Return the result when it arrives. If `timeout` is not `None` and the result does not arrive within `timeout` seconds then `TimeoutError` is raised.

Table 15.1: All information from `https://ipyparallel.readthedocs.io/en/latest/details.html#AsyncResult`.

There are additional magic methods supplied by `iPyParallel` that make some of these operations easier. These methods are contained in the Additional Material section. More information on `iPyParallel` architecture, interface, and methods at `https://ipyparallel.readthedocs.io/en/latest/index.html`.

Applications

Parallel computing, when used correctly, is one of the best ways to speed up the run time of an algorithm. As a result, it is very commonly used today and has many applications, such as the following:

- Graphic rendering
- Facial recognition with large databases
- Numerical integration
- Calculating Discrete Fourier Transforms
- Simulation of various natural processes (weather, genetics, etc.)
- Natural language processing

In fact, there are many problems that are only possible to solve through parallel computing because solving them serially would take too long. In these types of problems, even the parallel solution could take years. Some brute-force algorithms, like those used to crack simple encryptions, are examples of this type of problem.

The problems mentioned above are well suited to parallel computing because they can be manipulated in such a way that running them on multiple processors results in a significant run time improvement. Manipulating an algorithm to be run with parallel computing is called *parallelizing* the algorithm. When a problem only requires very minor manipulations to parallelize, it is often called *embarrassingly parallel*. Typically, an algorithm is embarrassingly parallel when there is little to no dependency between results. Algorithms that do not meet this criteria can still be parallelized, but there is not always a significant enough improvement in run time to make it worthwhile. For example, calculating the Fibonacci sequence using the usual formula, $\mathrm{F}(n) = \mathrm{F}(n-1) + \mathrm{F}(n-2)$, is poorly suited to parallel computing because each element of the sequence is dependent on the previous two elements.

Problem 5. The *trapeziod rule* is a simple technique for numerical integration:

$$\int_a^b f(x)dx \approx \frac{h}{2}\sum_{k=1}^{N}(f(x_k) + f(x_{k+1})),$$

where $a = x_1 < x_2 < \ldots < x_N = b$ and $h = x_{n+1} - x_n$ for each n. See Figure 15.2.

Note that estimation of the area of each interval is independent of all other intervals. As a result, this problem is considered embarrassingly parallel.

Write a function that accepts a function handle to integrate, bounds of integration, and the number of points to use for the approximation. Parallelize the trapezoid rule in order to estimate the integral of f. That is, evenly divide the points among all available processors and run the trapezoid rule on each portion simultaneously. The sum of the results of all the processors will be the estimation of the integral over the entire interval of integration. Return this sum.

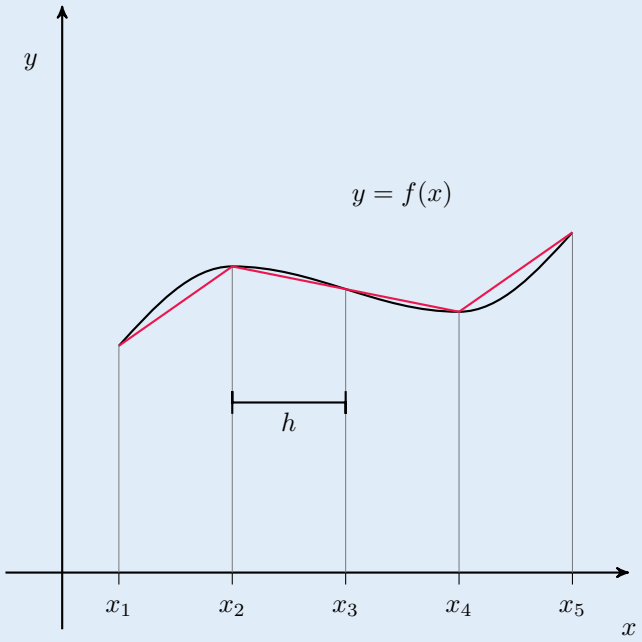


Figure 15.2: A depiction of the trapezoid rule with uniform partitioning.

Intercommunication

The phrase *parallel computing* refers to designing an architecture and code that makes the best use of computing resources for a problem. Occasionally, this will require nodes to be interdependent on each other for previous results. This contributes to a slower result because it requires a great deal of communication latency, but is sometimes the only method to parallelize a function. Although important, the ability to effectively communicate between engines has not been added to `iPyParallel`. It is, however, possible in an MPI framework and will be covered in a later lab.

Additional Material

Installation and Initialization

If you have not already installed `ipyparallel`, you may do so using the conda package manager.

```
$ conda update conda
$ conda update anaconda
$ conda install ipyparallel
```

Clusters of Multiple Machines

Though setting up a computing cluster with `iPyParallel` on multiple machines is similar to a cluster on a single computer, there are a couple of extra considerations to make. The majority of these considerations have to do with the network setup of your machines, which is unique to each situation. However, some basic steps have been taken from `https://ipyparallel.readthedocs.io/en/latest/process.html` and are outlined below.

SSH Connection

When using engines and controllers that are on separate machines, their communication will most likely be using an SSH tunnel. This *Secure Shell* allows messages to be passed over the network.

In order to enable this, an SSH user and IP address must be established when starting the controller. An example of this follows.

```
$ ipcontroller --ip=<controller IP> --user=<user of controller> --enginessh=<↩
    user of controller>@<controller IP>
```

Engines started on remote machines then follow a similar format.

```
$ ipengine --location=<controller IP> --ssh=<user of controller>@<controller IP↩
    >
```

Another way of affecting this is to alter the configuration file in `UserDirectory/.ipython/profile-default/security/ipcontroller-engine.json`. This can be modified to contain the controller IP address and SSH information.

All of this is dependent on the network feasibility of SSH connections. If there are a great deal of remote engines, this method will also require the SSH password to be entered many times. In order to avoid this, the use of SSH Keys from computer to computer is recommended.

Magic Methods & Decorators

To be more usable, the `iPyParallel` module has incorporated a few magic methods and decorators for use in an interactive iPython or Python terminal.

Magic Methods

The `iPyParallel` module has a few magic methods that are very useful for quick commands in iPython or in a Jupyter Notebook. The most important are as follows. Additional methods are found at `https://ipyparallel.readthedocs.io/en/latest/magics.html`.

%px - This magic method runs the corresponding Python command on the engines specified in `dview.targets`.

%autopx - This magic method enables a boolean that runs any code run on every engine until `%autopx` is run again.

Examples of these magic methods with a client and four engines are as follows.

```
# %px
In [4]: with dview.sync_imports():
   ...:     import numpy
   ...:
importing numpy on engine(s)
In [5]: \%px a = numpy.random.random(2)

In [6]: dview['a']
Out[6]:
[array([ 0.30390162,  0.14667075]),
 array([ 0.95797678,  0.59487915]),
 array([ 0.20123566,  0.57919846]),
 array([ 0.87991814,  0.31579495])]

 # %autopx
In [7]: %autopx
%autopx enabled
In [8]: max_draw = numpy.max(a)

In [9]: print('Max_Draw: {}'.format(max_draw))
[stdout:0] Max_Draw: 0.30390161663280246
[stdout:1] Max_Draw: 0.957976784975849
[stdout:2] Max_Draw: 0.5791984571339429
[stdout:3] Max_Draw: 0.8799181411958089

In [10]: %autopx
%autopx disabled
```

Decorators

The `iPyParallel` module also has a few decorators that are very useful for quick commands. The two most important are as follows:

@remote - This decorator creates methods on the remote engines.

@parallel - This decorator creates methods on remote engines that break up element wise operations and recombine results.

Examples of these decorators are as follows.

```
# Remote decorator
>>> @dview.remote(block=True)
>>> def plusone():
...     return a+1
>>> dview['a'] = 5
>>> plusone()
 [6, 6, 6, 6,]

# Parallel decorator
>>> import numpy as np

>>> @dview.parallel(block=True)
>>> def combine(A,B):
...     return A+B
>>> ex1 = np.random.random((3,3))
>>> ex2 = np.random.random((3,3))
>>> print(ex1+ex2)
 [[ 0.87361929  1.41110357  0.77616724]
 [ 1.32206426  1.48864976  1.07324298]
 [ 0.6510846   0.45323311  0.71139272]]
>>> print(combine(ex1,ex2))
 [[ 0.87361929  1.41110357  0.77616724]
 [ 1.32206426  1.48864976  1.07324298]
 [ 0.6510846   0.45323311  0.71139272]]
```

Connecting `iPyParallel` with MPI

The `iPyParallel` cluster can be imbued with the ability to interpret MPI commands. More information on making this connection can be found at `https://ipyparallel.readthedocs.io/en/latest/mpi.html`.

16 Parallel Programming with MPI

Lab Objective: *In the world of parallel computing, MPI is the most widespread and standardized message passing library. As such, it is used in the majority of parallel computing programs. In this lab, we explore and practice the basic principles and commands of MPI to further recognize when and how parallelization can occur.*

MPI: the Message Passing Interface

At its most basic, the Message Passing Interface (MPI) provides functions for sending and receiving messages between different processes. MPI was developed to provide a standard framework for parallel computing in any language. It specifies a library of functions — the syntax and semantics of message passing routines — that can be called from programming languages such as Fortran and C.

MPI can be thought of as "the assembly language of parallel computing," because of this generality.[1] MPI is important because it was the first portable and universally available standard for programming parallel systems and continues to be the de facto standard today.

> **Note**
>
> Most modern personal computers now have multicore processors. Programs that are designed for these multicore processors are "parallel" programs and are typically written using OpenMP or POSIX threads. MPI, on the other hand, is designed for any general architecture.

Why MPI for Python?

In general, programming in parallel is more difficult than programming in serial because it requires managing multiple processors and their interactions. Python, however, is an excellent language for simplifying algorithm design because it allows for problem solving without too much detail. Unfortunately, Python is not designed for high performance computing and is a notably slower scripted language. It is best practice to prototype in Python and then to write production code in fast compiled languages such as C or Fortran.

[1] *Parallel Programming with MPI*, by Peter S. Pacheco, pg. 7.

In this lab, we will explore the Python library `mpi4py` which retains most of the functionality of C implementations of MPI and is a good learning tool. If you do not have the MPI library and mpi4py installed on your machine, please refer to the Additional Material at the end of this lab. There are three main differences to keep in mind between mpi4py and MPI in C:

- Python is array-based while C is not.
- mpi4py is object oriented but MPI in C is not.
- mpi4py supports two methods of communication to implement each of the basic MPI commands. They are the upper and lower case commands (e.g. `Bcast(...)` and `bcast(...)`). The uppercase implementations use traditional MPI datatypes while the lower case use Python's pickling method. Pickling offers extra convenience to using mpi4py, but the traditional method is faster. In these labs, we will only use the uppercase functions.

Using MPI

We will start with a Hello World program.

```
#hello.py
from mpi4py import MPI

COMM = MPI.COMM_WORLD
RANK = COMM.Get_rank()

print("Hello world! I'm process number {}.".format(RANK))
```

hello.py

Save this program as `hello.py` and execute it from the command line as follows:

```
$ mpiexec -n 5 python hello.py
```

The program should output something like this:

```
Hello world! I'm process number 3.
Hello world! I'm process number 2.
Hello world! I'm process number 0.
Hello world! I'm process number 4.
Hello world! I'm process number 1.
```

Notice that when you try this on your own, the lines will not necessarily print in order. This is because there will be five separate processes running autonomously, and we cannot know beforehand which one will execute its `print()` statement first.

Achtung!

It is usually bad practice to perform I/O (e.g., call `print()`) from any process besides the root process (rank 0), though it can be a useful tool for debugging.

How does this program work? First, the `mpiexec` program is launched. This is the program which starts MPI, a wrapper around whatever program you to pass into it. The `-n 5` option specifies the desired number of processes. In our case, 5 processes are run, with each one being an instance of the program "python". To each of the 5 instances of python, we pass the argument `hello.py` which is the name of our program's text file, located in the current directory. Each of the five instances of python then opens the `hello.py` file and runs the same program. The difference in each process's execution environment is that the processes are given different ranks in the communicator. Because of this, each process prints a different number when it executes.

MPI and Python combine to make succinct source code. In the above program, the line `from mpi4py import MPI` loads the MPI module from the mpi4py package. The line `COMM = MPI.COMM_WORLD` accesses a static communicator object, which represents a group of processes which can communicate with each other via MPI commands. The next line, `RANK = COMM.Get_rank()`, accesses the processes *rank* number. A rank is the process's unique ID within a communicator, and they are essential to learning about other processes. When the program `mpiexec` is first executed, it creates a global communicator and stores it in the variable `MPI.COMM_WORLD`. One of the main purposes of this communicator is to give each of the five processes a unique identifier, or rank. When each process calls `COMM.Get_rank()`, the communicator returns the rank of that process. `RANK` points to a local variable, which is unique for every calling process because each process has its own separate copy of local variables. This gives us a way to distinguish different processes while writing all of the source code for the five processes in a single file.

Here is the syntax for `Get_size()` and `Get_rank()`, where `Comm` is a communicator object:

Comm.Get_size() Returns the number of processes in the communicator. It will return the same number to every process. Parameters:

Return value - the number of processes in the communicator

Return type - integer

Example:

```
#Get_size_example.py
from mpi4py import MPI
SIZE = MPI.COMM_WORLD.Get_size()
print("The number of processes is {}.".format(SIZE))
```

Get_size_example.py

Comm.Get_rank() Determines the rank of the calling process in the communicator. Parameters:

Return value - rank of the calling process in the communicator

Return type - integer

Example:

```
#Get_rank_example.py
from mpi4py import MPI
```

```
RANK = MPI.COMM_WORLD.Get_rank()
print("My rank is {}.".format(RANK))
```

Get_rank_example.py

The Communicator

A communicator is a logical unit that defines which processes are allowed to send and receive messages. In most of our programs we will only deal with the `MPI.COMM_WORLD` communicator, which contains all of the running processes. In more advanced MPI programs, you can create custom communicators to group only a small subset of the processes together. This allows processes to be part of multiple communicators at any given time. By organizing processes this way, MPI can physically rearrange which processes are assigned to which CPUs and optimize your program for speed. Note that within two different communicators, the same process will most likely have a different rank.

Note that one of the main differences between `mpi4py` and MPI in C or Fortran, besides being array-based, is that `mpi4py` is largely object oriented. Because of this, there are some minor changes between the `mpi4py` implementation of MPI and the official MPI specification.

For instance, the MPI Communicator in `mpi4py` is a Python class and MPI functions like `Get_size()` or `Get_rank()` are instance methods of the communicator class. Throughout these MPI labs, you will see functions like `Get_rank()` presented as `Comm.Get_rank()` where it is implied that `Comm` is a communicator object.

Separate Codes in One File

When an MPI program is run, each process receives the same code. However, each process is assigned a different rank, allowing us to specify separate behaviors for each process. In the following code, the three processes perform different operations on the same pair of numbers.

```
#separateCode.py
from mpi4py import MPI
RANK = MPI.COMM_WORLD.Get_rank()

a = 2
b = 3
if RANK == 0:
  print a + b
elif RANK == 1:
  print a*b
elif RANK == 2:
  print max(a, b)
```

separateCode.py

Problem 1. Write a program in which processes with an even rank print "Hello" and process with an odd rank print "Goodbye." Print the process number along with the "Hello" or "Goodbye" (for example, "Goodbye from process 3").

Message Passing between Processes

Let us begin by demonstrating a program designed for two processes. One will draw a random number and then send it to the other. We will do this using the routines `Comm.Send()` and `Comm.Recv()`.

```
#passValue.py
import numpy as np
from mpi4py import MPI

COMM = MPI.COMM_WORLD
RANK = COMM.Get_rank()

if RANK == 1:  # This process chooses and sends a random value
    num_buffer = np.random.rand(1)
    print("Process 1: Sending: {} to process 0.".format(num_buffer))
    COMM.Send(num_buffer, dest=0)
    print("Process 1: Message sent.")
if RANK == 0:  # This process recieves a value from process 1
    num_buffer = np.zeros(1)
    print("Process 0: Waiting for the message... current num_buffer={}.".format↩
        (num_buffer))
    COMM.Recv(num_buffer, source=1)
    print("Process 0: Message recieved! num_buffer={}.".format(num_buffer))
```

passValue.py

To illustrate simple message passing, we have one process choose a random number and then pass it to the other. Inside the receiving process, we have it print out the value of the variable `num_buffer` before it calls `Recv()` to prove that it really is receiving the variable through the message passing interface.

Here is the syntax for `Send()` and `Recv()`, where `Comm` is a communicator object:

Comm.Send(buf, dest=0, tag=0) Performs a basic send from one process to another. Parameters:

buf (array-like) : data to send

dest (integer) : rank of destination

tag (integer) : message tag

The `buf` object is not as simple as it appears. It must contain a pointer to a Numpy array. It cannot, for example, simply pass a string. The string would have to be packaged inside an array first.

Comm.Recv(buf, source=0, tag=0, Status status=None) Basic point-to-point receive of data. Parameters:

buf (array-like) : initial address of receive buffer (choose receipt location)

source (integer) : rank of source

tag (integer) : message tag

status (Status) : status of object

Example:

```
#Send_example.py
from mpi4py import MPI
import numpy as np

RANK = MPI.COMM_WORLD.Get_rank()

a = np.zeros(1, dtype=int)  # This must be an array.
if RANK == 0:
    a[0] = 10110100
    MPI.COMM_WORLD.Send(a, dest=1)
elif RANK == 1:
    MPI.COMM_WORLD.Recv(a, source=0)
    print(a[0])
```

Send_example.py

Problem 2. Write a script that runs on two processes and passes an n by 1 vector of random values from one process to the other. Write it so that the user passes the value of n in as a command-line argument. The following code demonstrates how to access command-line arguments.

```
from sys import argv

# Pass in the first command line argument as n.
n = int(argv[1])
```

NOTE

`Send()` and `Recv()` are referred to as *blocking* functions. That is, if a process calls `Recv()`, it will sit idle until it has received a message from a corresponding `Send()` before it will proceed. (However, in Python the process that calls `Comm.Send` will *not* necessarily block until the message is received, though in C, `MPI_Send` does block) There are corresponding *non-blocking* functions `Isend()` and `Irecv()` (The *I* stands for immediate). In essence, `Irecv()` will return immediately. If a process calls `Irecv()` and doesn't find a message ready to be picked up, it will indicate to the system that it is expecting a message, proceed beyond the `Irecv()` to do other useful work, and then check back later to see if the message has arrived. This can be used to dramatically improve performance.

Problem 3. Write a script in which the process with rank i sends a random value to the process with rank $i+1$ in the global communicator. The process with the highest rank will send its random value to the root process. Notice that we are communicating in a ring. For communication, only use `Send()` and `Recv()`. The program should work for any number of processes. Does the order in which `Send()` and `Recv()` are called matter?

NOTE

When calling `Comm.Recv`, you can allow the calling process to accept a message from any process that happened to be sending to the receiving process. This is done by setting source to a predefined MPI constant, `source=ANY_SOURCE` (note that you would first need to import this with from `mpi4py.MPI import ANY_SOURCE` or use the syntax `source=MPI.ANY_SOURCE`).

Application: Monte Carlo Integration

Monte Carlo integration uses random sampling to approximate volumes (whereas most numerical integration methods employ some sort of regular grid). It is a useful technique, especially when working with higher-dimensional integrals. It is also well-suited to parallelization because it involves a large number of independent operations. In fact, Monte Carlo algorithms can be made "embarassingly parallel" — the processes don't need to communicate with one another during execution, simply reporting results to the root process upon completion.

In a simple example, the following code calculates the value of π by sampling random points inside the square $[-1,1] \times [-1,1]$. Since the volume of the unit circle is π and the volume of the square is 4, the probability of a given point landing inside the unit circle is $\pi/4$, so the proportion of samples that fall within the unit circle should also be $\pi/4$. The program samples $N = 2000$ points, determines which samples are within the unit circle (say M are), and estimates $\pi \approx 4M/N$.

```
# pi.py
import numpy as np
from scipy import linalg as la


# Get 2000 random points in the 2-D domain [-1,1]x[-1,1].
```

```
points = np.random.uniform(-1, 1, (2,2000))

# Determine how many points are within the unit circle.
lengths = la.norm(points, axis=0)
num_within = np.count_nonzero(lengths < 1)

# Estimate the circle's area.
print(4 * (num_within / 2000))
```

pi.py

```
$ python pi.py
3.166
```

Problem 4. The n-dimensional *open unit ball* is the set $U_n = \{\mathbf{x} \in \mathbb{R}^n \mid \|\mathbf{x}\|_2 < 1\}$. Write a script that accepts integers n and N on the command line. Estimate the volume of U_n by drawing N points over the n-dimensional domain $[-1,1] \times [-1,1] \times \cdots \times [-1,1]$ on each available process except the root process (for a total of $(r-1)N$ draws, where r is the number of processes). Have the root process print the volume estimate.
(Hint: the volume of $[-1,1] \times [-1,1] \times \cdots \times [-1,1]$ is 2^n.)

When $n = 2$, this is the same experiment outlined above so your function should return an approximation of π. The volume of the U_3 is $\frac{4}{3}\pi \approx 4.18879$, and the volume of U_4 is $\frac{\pi^2}{2} \approx 4.9348$. Try increasing the number of sample points N or processes r to see if your estimates improve.

NOTE

Good parallel code should pass as little data as possible between processes. Sending large or frequent messages requires a level of synchronization and causes some processes to pause as they wait to receive or send messages, negating the advantages of parallelism. It is also important to divide work evenly between simultaneous processes, as a program can only be as fast as its slowest process. This is called load balancing, and can be difficult in more complex algorithms.

Additional Material

Installing mpi4py

1. For All Systems: The easiest installation is using `conda install mpi4py`. You may also run `pip install mpi4py`

17 Introduction to Apache Spark

Lab Objective: *Being able to reasonably deal with massive amounts of data often requires parallelization and cluster computing. Apache Spark is an industry standard for working with big data. In this lab we introduce the basics of PySpark, Spark's Python API, including data structures, syntax, and use cases.*

Apache Spark

Apache Spark is an open-source, general-purpose distributed computing system used for big data analytics. Spark is able to complete jobs substantially faster than previous big data tools (i.e. Apache Hadoop) because of its in-memory caching, and optimized query execution. Spark provides development APIs in Python, Java, Scala, and R. On top of the main computing framework, Spark provides machine learning, SQL, graph analysis, and streaming libraries.

Spark's Python API can be accessed through the PySpark module. Installation for local execution or remote connection to an existing cluster can be easily done with `conda` or `pip` commands.

```
# PySpark installation with conda
>>> conda install -c conda-forge pyspark

# PySpark installation with pip
>>> pip install pyspark
```

Achtung!

If you need to setup Spark as a standalone cluster, using `conda` and `pip` is insufficient; instead you will need to use the PySpark prebuilt binaries. However, it is usually unnecessary to install PySpark using the prebuilt binaries.

Resilient Distributed Datasets and DataFrames

The main data structure used in Apache Spark is a Reslient Distributed Dataset (RDD). RDDs are immutable distributed collections of objects. Since RDDs are distributed, only a portion of the data is stored on each node.

Though RDDs are the main object that Spark operates on, they can be difficult to work with directly; instead, Spark offers another data structure, called a DataFrame, which is conceptually similar to a relational database. It is easy to convert DataFrames to RDDs when greater control is needed by calling the `.rdd` method on the DataFrame object. The reverse conversion can be done by calling `spark.createDataFrame()` on an existing RDD.

PySpark

One of the major benefits of using PySpark is the interactive shell which functions much like IPython. To use the shell, simply run `pyspark` in the terminal. In the Spark shell you can run code one line at a time without the need to have a fully written program. This is a great way to get a feel for Spark. To get help with a function us `help(function)`; to exit the shell simply run `quit()`.

```
>>> pyspark
Python 3.6.5 |Anaconda, Inc.| (default, Apr 29 2018, 16:14:56)
[GCC 7.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.3.1
      /_/

Using Python version 3.6.5 (default, Apr 29 2018 16:14:56)
SparkSession available as 'spark'.

>>> spark.read.text("my_text_file.txt").show(3)
+--------------------+
|               value|
+--------------------+
|One does not simp...|
|Its black gates a...|
|There is evil the...|
+--------------------+
only showing top 3 rows

>>> quit()
```

Using Spark in a Python script requires the `PySpark` module. For most cases you will also want to import `SparkSession` from the `pyspark.sql` submodule. To create a connection to and interact with the cluster you will need to instantiate `SparkContext` and `SparkSession` objects, respectively. It is standard to call your `SparkContext sc` and `SparkSession spark`; we will use these naming conventions throughout the remainder of the lab. It is important to note that when you are finished with a `SparkSession` you should end it by calling `spark.stop()`.

Note

When running Spark in the interactive shell `SparkSession` is available as `spark` by default. Furthermore, you don't need to worry about stopping the session when you `quit()`.

If you prefer a different interactive environment, like IPython, you just need to use the code given below. Help can be accessed in the usual way for your environment. Just remember to `stop()` the `SparkSession`!

```
>>> import pyspark
>>> from pyspark.sql import SparkSession

# Establish a connection to the cluster
>>> sc = pyspark.context.SparkContext()

# Instantiate your SparkSession object
>>> spark = SparkSession\
            .builder\
            .getOrCreate()

# Stop your SparkSession
>>> spark.stop()
```

Note

The syntax

```
>>> spark = SparkSession\
            .builder\
            .getOrCreate()
```

is somewhat unusual. While this code can be written on a single line, it is generally more readable to break it up when dealing with many chained operations. It is important to note that you *cannot* write a comment after a line continuation character \.

Spark SQL and DataFrames

Creating new DataFrame objects from text, csv, JSON, and other files can be done easily with the `spark.read()` method. If the DataFrame schema is specified on the first line of the document, use `spark.read.option("header", True)`. Additionally, you can create DataFrames from existing Pandas DataFrames, RDDs, numpy arrays, and lists with `spark.createDataFrame()`.

```
# SparkSession available as spark
>>> txt_df = spark.read.text("my_text_file.txt")        # text files
>>> csv_df = spark.read.csv("my_csv_file.csv")          # csv files
>>> json_df = spark.read.json("my_json.json")           # JSON files

# to use the document's first line as the schema
>>> txt_df_schema = spark.read.option("header", True).text("my_text_file")

# for Pandas DataFrames, RDDs, numpy arrays, etc.
>>> df_convert = spark.createDataFrame("my_data.npy")
```

The `spark.sql` module allows you to perform SQL operations on DataFrame objects. This can be incredibly useful when coupled with other Spark functions since you can update, query, and analyze data in a single, unified engine. As previously mentioned, DataFrame objects can be generally regarded as functioning in the same way as a relational database.

While many SQL operations found in the Spark SQL module share the same name, there are some that differ. The main difference between standard SQL and Spark SQL in PySpark is the syntax; given a DataFrame object `df`, to select a column, for example, you would type: `df.select("col_name")` or `df.select(df.col_name)`.

```
# SparkSession available as spark
>>> df.select("name").show(3) # equivalent to df.select(df.name)
+--------+
|    Name|
+--------+
|   Sarah|
|    Andy|
|   Kevin|
+--------+
only showing top 3 rows
```

Spark SQL Command	SQLite Command
`select(*cols)`	`SELECT`
`groupBy(*cols)`	`GROUP BY`
`sort(*cols, **kwargs)`	`ORDER BY`
`filter(condition)`	`WHERE`
`when(condition, value)`	`WHEN`
`between(lowerBound, upperBound)`	`BETWEEN`
`count()`	`COUNT()`
`collect()`	`fetchall()`

Problem 1. Write a function that accepts the file `mathematicians.csv`, which contains basic data on over 8000 mathematicians throughout history, and use it to create a Spark DataFrame. [a] Filter this DataFrame to contain only the names of female mathematicians born in the 19th century (1801-1900). Return a list containing the first 5 names.

The following may be useful for extracting the names from the DataFrame `row` objects:

```
# assuming df_names is a single column DataFrame of the desired names
>>> df_names.rdd \
...          .flatMap(lambda x: x) \
...          .collect()[:5]
['First Name', 'Second Name', 'Third Name', 'Fourth Name', 'Fifth Name']
```

[a] https://www.kaggle.com/joephilleo/mathematicians-on-wikipedia

Problem 2. Write a function that accepts the file `mathematicians.csv` and use it to create a Spark DataFrame. Query the DataFrame to count the number of mathematicians belonging to each country. Sort the countries by count in descending order. Return a list of the top 5 `(country, count)` tuples.

The following may be useful for extracting the `(country, count)` tuples:

```
# assuming country_count is a DataFrame with schema (country, count)
>>> country_count.rdd \
...               .map(lambda x: x[:2]) \
...               .collect()[:2]
[('First Country', count_1), ('Second Country', count_2)]
```

RDDs

There are two main operations that you perform on RDDs in Spark: `map()` and `reduce()`. The `map(f)` method returns a new RDD by applying a function, `f`, to each element of the original RDD. The `reduce(f)` method reduces the data using the specified commutative and associative binary operator, `f`. The function, `f`, for `map(f)` and the binary operator for `reduce(f)` is often specified using a `lambda` funtion.

```
# create an RDD from a text file
>>> my_data = spark.read.text("my_text_file.txt").rdd

>>> my_data.first()     # display the first element of the RDD
Row(value='One does not simply walk into Mordor.')  # returns a Row object

>>> my_data.map(lambda r: r[0]).first() # extract content from the Row object
'One does not simply walk into Mordor.'
```

```
# combine each line, returning the whole document as a single string
>>> my_data.map(lambda r: r[0]).reduce(lambda a, b: a + " " + b)
'One does not simply walk into Mordor. Its Black Gates are guarded by more than↩
    just Orcs. There is evil there that does not sleep, and the Great Eye is ↩
   ever watchful.'
```

Problem 3. Write a function that accepts the name of a text file. Create a `SparkSession`, load the file as a DataFrame, convert it to an RDD, count the number of occurences of each word, and sort the words by count in descending order. Return a list of tuples containing the first five `(word, count)` pairs.

Hint: If you have an RDD containing the lines of the file, what does `lines.flatMap(lambda x: x.split(" "))` do? Also consider using `reduceByKey()`.

One way to create RDDs that are ready for parallel computing is to use `sc.parallelize(c, numSlices=None)` (recall that `sc` is an instance of the `SparkContext` object). This will partition a local Python collection, `c`. Each partition can then be sent to a separate node for processing. The `numSlices` keyword argument specifies the number of partitions to create. Combining this with `range(n)` provides an efficient way to distribute and run a specific `map()` process `n` times.

```
import numpy as np
# SparkContext available as sc

# a Python collection we wish to parallelize
>>> c = ['a', 'b', 'c', 'd']
>>> sc.parallelize(c, 2).glom().collect()
[['a', 'b'], ['c', 'd']]
>>> sc.parallelize(c, 4).glom().collect()
[['a'], ['b'], ['c'], ['d']]

# simulate flipping a coin 10 times; x is a dummy variable
>>> toss = sc.parallelize(range(10), 2).map(lambda x: np.random.randint(2))
>>> toss.collect()
[0, 0, 1, 1, 1, 0, 0, 1, 1, 1]
>>> toss.reduce(lambda a, b: a + b) # reduce by summing the entries
6
```

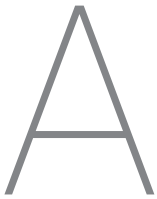
Problem 4. Since the area of a circle of radius r is $A = \pi r^2$, one way to estimate π is to estimate the area of the unit circle. A Monte Carlo approach to this problem is to uniformly sample points in the square $[-1, 1] \times [-1, 1]$ and then count the percentage of points that land within the unit circle. The percentage of points within the circle approximates the percentage of the area occupied by the circle. Multiplying this percentage by 4 (the area of the square $[-1, 1] \times [-1, 1]$) gives an estimate for the area of the circle. [a]

Write a function that uses Monte Carlo methods to estimate the value of π. Use Spark's `parallelize()` method to create a partitioned RDD with n entries, where $n = (10^5 * \text{partitions})$. Specify a keyword argument, `partitions=2`, to control the number of partitions for the RDD. Use `map()` and `reduce()` operations on the RDD to estimate π.

[a] See Example 7.1.1 in the Volume 2 textbook

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`
- 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>"`.

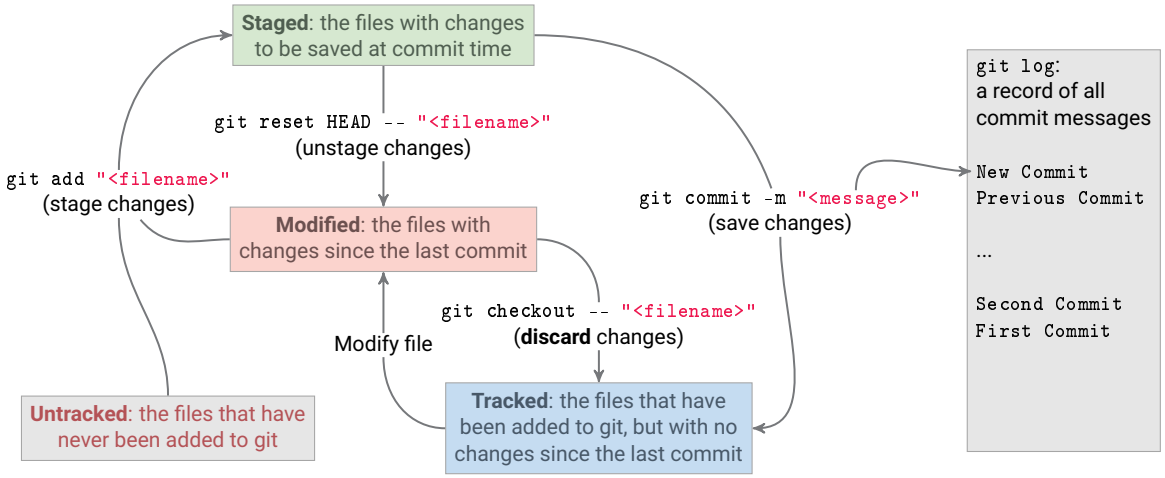


Figure A.1: Git commands to stage, unstage, save, or discard changes. Commit messages are recorded in the log.

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.

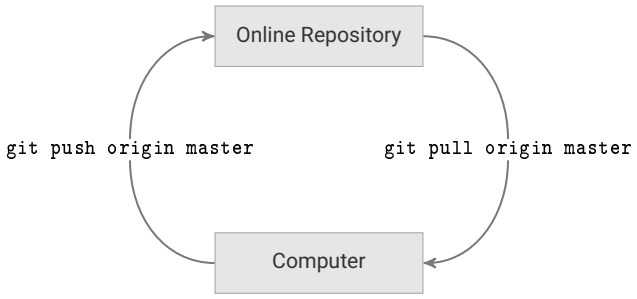


Figure A.2: Exchanging git commits between the repository and a local clone.

Command	Explanation
`git status`	Display the staging area and untracked changes.
`git pull origin master`	Pull changes from the online repository.
`git push origin master`	Push changes to the online repository.
`git add <filename(s)>`	Add a file or files to the staging area.
`git add -u`	Add all modified, tracked files to the staging area.
`git commit -m "<message>"`	Save the changes in the staging area with a given message.
`git checkout -- <filename>`	Revert changes to an unstaged file since the last commit.
`git reset HEAD -- <filename>`	Remove a file from the staging area.
`git diff <filename>`	See the changes to an unstaged file since the last commit.
`git diff --cached <filename>`	See the changes to a staged file since the last commit.
`git config --local <option>`	Record your credentials (`user.name`, `user.email`, 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)`), 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 Sessions

```
$ cd ~/Desktop/Student-Materials/
$ git pull origin master                          # Pull updates.
### Make changes to a file.
$ git add -u                                      # Track changes.
$ git commit -m "Made some changes."              # Commit changes.
$ git push origin master                          # Push updates.
```

```
# Pull any updates from the online repository (such as TA feedback).
$ cd ~/Desktop/Student-Materials/
$ 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

$ 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
`conda install <package-name>`	Install the specified package.
`conda update <package-name>`	Update the specified package.
`conda update conda`	Update `conda` itself.
`conda update anaconda`	Update **all** packages included in Anaconda.
`conda --help`	Display the documentation for `conda`.

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
`pip install package-name`	Install the specified package.
`pip install --upgrade package-name`	Update the specified package.
`pip freeze`	Display the version number on all installed packages.
`pip --help`	Display the documentation for `pip`.

See `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/`.
> - 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, Juptyer 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 LaTeX, and can embedded 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.

C NumPy Visual Guide

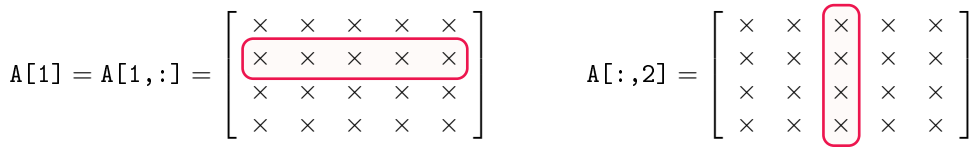
Lab Objective: *NumPy operations can be difficult to visualize, but the concepts are straightforward. This appendix provides visual demonstrations of how NumPy arrays are used with slicing syntax, stacking, broadcasting, and axis-specific operations. Though these visualizations are for 1- or 2-dimensional arrays, the concepts can be extended to n-dimensional arrays.*

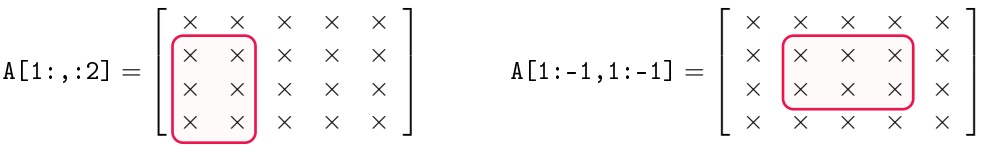
Data Access

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.

Slicing

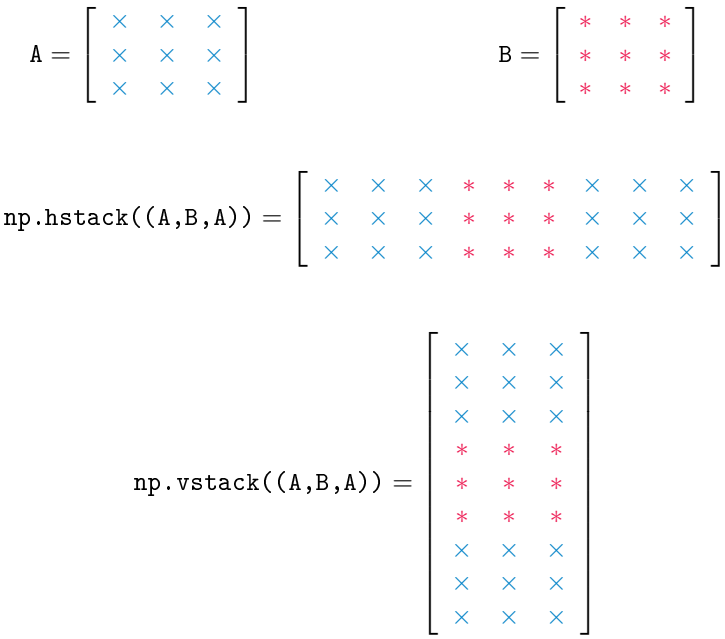
A lone colon extracts an entire row or column from a 2-D array. The syntax `[a:b]` can be read as "the ath entry up to (but not including) the bth entry." Similarly, `[a:]` means "the ath entry to the end" and `[:b]` means "everything up to (but not including) the bth entry."



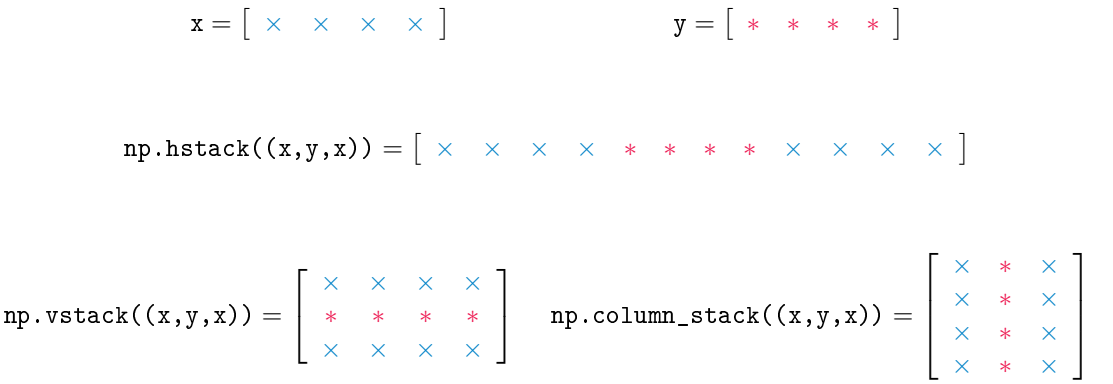


Stacking

`np.hstack()` stacks sequence of arrays horizontally and `np.vstack()` stacks a sequence of arrays vertically.



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()`.



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.

$$\texttt{A} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} \qquad \texttt{x} = \begin{bmatrix} 10 & 20 & 30 \end{bmatrix}$$

$$\texttt{A + x} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 10 & 20 & 30 \end{bmatrix} = \begin{bmatrix} 11 & 22 & 33 \\ 11 & 22 & 33 \\ 11 & 22 & 33 \end{bmatrix}$$

$$\texttt{A + x.reshape((1,-1))} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix} = \begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{bmatrix}$$

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`.

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

$$\texttt{A.sum(axis=0)} = \begin{bmatrix} \boxed{1} & \boxed{2} & \boxed{3} & \boxed{4} \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} = \begin{bmatrix} 4 & 8 & 12 & 16 \end{bmatrix}$$

$$\texttt{A.sum(axis=1)} = \begin{bmatrix} \boxed{1 & 2 & 3 & 4} \\ \boxed{1 & 2 & 3 & 4} \\ \boxed{1 & 2 & 3 & 4} \\ \boxed{1 & 2 & 3 & 4} \end{bmatrix} = \begin{bmatrix} 10 & 10 & 10 & 10 \end{bmatrix}$$

D Introduction to Scikit-Learn

Lab Objective: *Scikit-learn is the one of the fundamental tools in Python for machine learning. In this appendix we highlight and give examples of some popular scikit-learn tools for classification and regression, training and testing, data normalization, and constructing complex models.*

> **Note**
>
> This guide corresponds to scikit-learn version 0.20, which has a few significant differences from previous releases. See `http://scikit-learn.org/stable/whats_new.html` for current release notes. Install scikit-learn (the `sklearn` module) with `conda install scikit-learn`.

Base Classes and API

Many machine learning problems center on constructing a function $f : X \to Y$, called a *model* or *estimator*, that accurately represents properties of given data. The domain X is usually $\mathbb{R}^D$, and the range Y is typically either $\mathbb{R}$ (regression) or a subset of $\mathbb{Z}$ (classification). The model is trained on N *samples* $(\mathbf{x}_i)_{i=1}^N \subset X$ that usually (but not always) have N accompanying *labels* $(y_i)_{i=1}^N \subset Y$.

Scikit-learn [PVG+11, BLB+13] takes a highly object-oriented approach to machine learning models. Every major scikit-learn class inherits from `sklearn.base.BaseEstimator` and conforms to the following conventions:

1. The constructor `__init__()` receives *hyperparameters* for the classifier, which are parameters for the model f that are **not dependent on data**. Each hyperparameter must have a default value (i.e., every argument of `__init__()` is a keyword argument), and each argument must be saved as an instance variable of the **same name** as the parameter.

2. The `fit()` method constructs the model f. It receives an $N \times D$ matrix X and, optionally, a vector $\mathbf{y}$ with N entries. Each row $\mathbf{x}_i$ of X is one sample with corresponding label y_i. By convention, `fit()` always returns `self`.

Along with the `BaseEstimator` class, there are several other "mix in" base classes in `sklearn.base` that define specific kinds of models. The three listed below are the most common.[1]

[1] See `http://scikit-learn.org/stable/modules/classes.html#base-classes` for the complete list.

- `ClassifierMixin`: for *classifiers*, estimators that take on discrete values.
- `RegressorMixin`: for *regressors*, estimators that take on continuous values.
- `TransformerMixin`: for preprocessing data before estimation.

Classifiers and Regressors

The `ClassifierMixin` and `RegressorMixin` both require a `predict()` method that acts as the actual model f. That is, `predict()` receives an $N \times D$ matrix X and returns N predicted labels $(y_i)_{i=1}^N$, where y_i is the label corresponding to the ith row of X. Both of these base class have a predefined `score()` method that uses `predict()` to test the accuracy of the model. It accepts $N \times D$ test data and a vector of N corresponding labels, then reports either the classification accuracy (for classifiers) or the R^2 value of the regression (for regressors).

For example, a `KNeighborsClassifier` from `sklearn.neighbors` inherits from `BaseEstimator` and `ClassifierMixin`. This classifier uses a simple strategy: to classify a new piece of data $\mathbf{z}$, find the k training samples that are "nearest" to $\mathbf{z}$, then take the most common label corresponding to those nearest neighbors to be the label for $\mathbf{z}$. Its constructor accepts hyperparameters such as `n_neighbors`, for determining the number of neighbors k to search for, `algorithm`, which speficies the strategy to find the neighbors, and `n_jobs`, the number of parallel jobs to run during the neighbors search. Again, these hyperparameters are independent of any data, which is why they are set in the constructor (before fitting the model). Calling `fit()` organizes the data X into a data structure for efficient nearest neighbor searches (determined by `algorithm`). Calling `predict()` executes the search, determines the most common label of the neighbors, and returns that label.

```
>>> from sklearn.datasets import load_breast_cancer
>>> from sklearn.neighbors import KNeighborsClassifier
>>> from sklearn.model_selection import train_test_split

# Load the breast cancer dataset and split it into training and testing groups.
>>> cancer = load_breast_cancer()
>>> X_train, X_test, y_train, y_test = train_test_split(cancer.data,
...                                                     cancer.target)
>>> print(X_train.shape, y_train.shape)
(426, 30) (426,)        # There are 426 training points, each with 30 features.

# Train a KNeighborsClassifier object on the training data.
# fit() returns the object, so we can instantiate and train in a single line.
>>> knn = KNeighborsClassifier(n_neighbors=2).fit(X_train, y_train)
# The hyperparameter 'n_neighbors' is saved as an attribute of the same name.
>>> knn.n_neighbors
2

# Test the classifier on the testing data.
>>> knn.predict(X_test[:6])
array([0, 1, 0, 1, 1, 0])       # Predicted labels for the first 6 test points.
>>> knn.score(X_test, y_test)
0.8951048951048951              # predict() chooses 89.51% of the labels right.
```

The `KNeighborsClassifier` object could easily be replaced with a different classifier, such as a `GaussianNB` object from `sklearn.naive_bayes`. Since `GaussianNB` also inherits from `BaseEstimator` and `ClassifierMixin`, it has `fit()`, `predict()`, and `score()` methods that take in the same kinds of inputs as the corresponding methods for the `KNeighborsClassifier`. The only difference, from an external perspective, is the hyperparameters that the constructor accepts.

```
>>> from sklearn.naive_bayes import GaussianNB

>>> gnb = GaussianNB().fit(X_train, y_train)
>>> gnb.predict(X_test[:6])
array([1, 1, 0, 1, 1, 0])
>>> gnb.score(X_test, y_test)
0.9440559440559441
```

Roughly speaking, the `GaussianNB` classifier assumes all features in the data are independent and normally distributed, then uses Bayes' rule to compute the likelihood of a new point belonging to a label for each of the possible labels. To do this, the `fit()` method computes the mean and variance of each feature, grouped by label. These quantities are saved as the attributes `theta_` (the means) and `sigma_` (the variances), then used in `predict()`. Parameters like these that **are dependent on data** are only defined in `fit()`, not the constructor, and they are always named with a trailing underscore. These "non-hyper" parameters are often simply called *model parameters*.

```
>>> gnb.classes_        # The collection of distinct training labels.
array([0, 1])
>>> gnb.theta_[:,0]     # The means of the first feature, grouped by label.
array([17.55785276, 12.0354981 ])
# The samples with label 0 have a mean of 17.56 in the first feature.
```

The `fit()` method should do all of the "heavy lifting" by calculating the model parameters. The `predict()` method should then use these parameters to choose a label for test data.

	Hyperparameters	Model Parameters
Data dependence	No	Yes
Initialization location	`__init__()`	`fit()`
Naming convention	Same as argument name	Ends with an underscore
Examples	`n_neighbors`, `algorithm`, `n_jobs`	`classes_`, `theta_`, `sigma_`

Table D.1: Naming and initialization conventions for scikit-learn model parameters.

Building Custom Estimators

The consistent conventions in the various scikit-learn classes makes it easy to use a wide variety of estimators with near-identical syntax. These conventions also makes it possible to write custom estimators that behave like native scikit-learn objects. This usually only involves writing `fit()` and `predict()` methods and inheriting from the appropriate base classes. As a simple (though poorly performing) example, consider an estimator that either always predicts the same user-provided label, or that always predicts the most common label in the training data. Which strategy to use is independent of the data, so we encode that behavior with hyperparameters; the most common label must be calculated from the data, so that is a model parameter.

```
>>> import numpy as np
>>> from collections import Counter
>>> from sklearn.base import BaseEstimator, ClassifierMixin

>>> class PopularClassifier(BaseEstimator, ClassifierMixin):
...     """Classifier that always guesses the most common training label."""
...     def __init__(self, strategy="most_frequent", constant=None):
...         self.strategy = strategy    # Store the hyperparameters, using
...         self.constant = constant    # the same names as the arguments.
...
...     def fit(self, X, y):
...         """Find and store the most common label."""
...         self.popular_label_ = Counter(y).most_common(1)[0][0]
...         return self                 # fit() always returns 'self'.
...
...     def predict(self, X):
...         """Always guess the most popular training label."""
...         M = X.shape[0]
...         if self.strategy == "most_frequent":
...             return np.full(M, self.popular_label_)
...         elif self.strategy == "constant":
...             return np.full(M, self.constant)
...         else:
...             raise ValueError("invalid value for 'strategy' param")
...
# Train a PopularClassifier on the breast cancer training data.
>>> pc = PopularClassifier().fit(X_train, y_train)
>>> pc.popular_label_
1
# Score the model on the testing data.
>>> pc.score(X_test, y_test)
0.6573426573426573                  # 65.73% of the testing data is labeled 1.

# Change the strategy to always guess 0 by changing the hyperparameters.
>>> pc.strategy = "constant"
>>> pc.constant = 0
>>> pc.score(X_test, y_test)
0.34265734265734266                  # 34.27% of the testing data is labeled 0.
```

This is a terrible classifier, but it is actually implemented as `sklearn.dummy.DummyClassifier` because any legitimate machine learning algorithm should be able to beat it, so it is useful as a baseline comparison.

Note that `score()` was inherited from `ClassifierMixin` (it isn't defined explicitly), so it returns a classification rate. In the next example, a slight simplification of the equally unintelligent `sklearn.dummy.DummyRegressor`, the `score()` method is inherited from `RegressorMixin`, so it returns an R^2 value.

```
>>> from sklearn.base import RegressorMixin

>>> class ConstRegressor(BaseEstimator, RegressorMixin):
...     """Regressor that always predicts a mean or median of training data."""
...     def __init__(self, strategy="mean", constant=None):
...         self.strategy = strategy    # Store the hyperparameters, using
...         self.constant = constant    # the same names as the arguments.
...
...     def fit(self, X, y):
...         self.mean_, self.median_ = np.mean(y), np.median(y)
...         return self                 # fit() always returns 'self'.
...
...     def predict(self, X):
...         """Always predict the middle of the training data."""
...         M = X.shape[0]
...         if self.strategy == "mean":
...             return np.full(M, self.mean_)
...         elif self.strategy == "median":
...             return np.full(M, self.median_)
...         elif self.strategy == "constant":
...             return np.full(M, self.constant)
...         else:
...             raise ValueError("invalid value for 'strategy' param")
...
# Train on the breast cancer data (treating it as a regression problem).
>>> cr = ConstRegressor(strategy="mean").fit(X_train, y_train)
>>> print("mean:", cr.mean_, " median:", cr.median_)
mean: 0.6173708920187794  median: 1.0

# Get the R^2 score of the regression on the testing data.
>>> cr.score(X_train, y_train)
0                           # Unsurprisingly, no correlation.
```

Achtung!

Both `PopularClassifier` and `ConstRegressor` wait until `predict()` to validate the `strategy` hyperparameter. The check could easily be done in the constructor, but that goes against scikit-learn conventions: in order to cooperate with automated validation tools, the constructor of any class inheriting from `BaseEstimator` must store the arguments of `__init__()` as attributes—with the same names as the arguments—and do nothing else.

Note

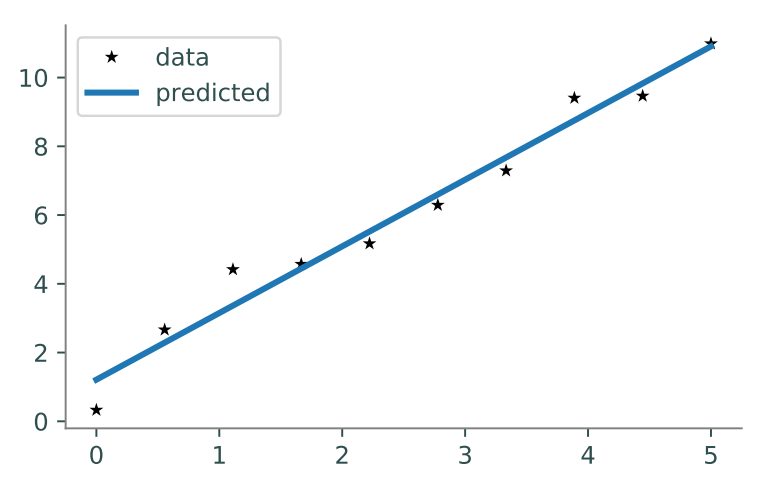
The first input to `fit()` and `predict()` are **always** two-dimensional $N \times D$ NumPy arrays, where N is the number of observations and D is the number of features. To fit or predict on one-dimensional data ($D = 1$), reshape the input array into a "column vector" before feeding it into the estimator. One-dimensional problems are somewhat rare in machine learning, but the following example shows how to do a simple one-dimensional linear regression.

```
>>> from matplotlib import pyplot as plt
>>> from sklearn.linear_model import LinearRegression

# Generate data for a 1-dimensional regression problem.
>>> X = np.linspace(0, 5, 10)
>>> Y = 2*X + 1 + np.random.normal(size=10)

# Reshape the training data into a column vector.
>>> lr = LinearRegression().fit(X.reshape((-1,1)), Y)

# Define another set of points to do predictions on.
>>> x = np.linspace(0, 5, 20)
>>> y = lr.predict(x.reshape((-1,1)))   # Reshape before predicting.
>>> plt.plot(X, Y, 'k*', label="data")
>>> plt.plot(x, y, label="predicted")
>>> plt.legend(loc="upper left")
>>> plt.show()
```



Transformers

A scikit-learn *transformer* processes data to make it better suited for estimation. This may involve shifting and scaling data, dropping columns, replacing missing values, and so on.

Classes that inherit from the `TransformerMixin` base class have a `fit()` method that accepts an $N \times D$ matrix X (like an estimator) and an optional set of labels. The labels are not needed—in fact the `fit()` method should do nothing with them—but the parameter for the labels remains as a keyword argument to be consistent with the `fit(X,y)` syntax of estimators. Instead of a `predict()` method, the `transform()` method accepts data, modifies it (usually via a copy), and returns the result. The new data may or may not have the same number of columns as the original data.

One common transformation is shifting and scaling the features (columns) so that they each have a mean of 0 and a standard deviation of 1. The following example implements a basic version of this transformer.

```
>>> from sklearn.base import TransformerMixin

>>> class NormalizingTransformer(BaseEstimator, TransformerMixin):
...     def fit(self, X, y=None):
...         """Calculate the mean and standard deviation of each column."""
...         self.mu_ = np.mean(X, axis=0)
...         self.sig_ = np.std(X, axis=0)
...         return self
...
...     def transform(self, X):
...         """Center each column at zero and normalize it."""
...         return (X - self.mu_) / self.sig_
...
# Fit the transformer and transform the cancer data (both train and test).
>>> nt = NormalizingTransformer()
>>> Z_train = nt.fit_transform(X_train) # Or nt.fit(X_train).transform(X_train)
>>> Z_test = nt.transform(X_test)       # Transform test data (without fitting)

>>> np.mean(Z_train, axis=0)[:3]        # The columns of Z_train have mean 0...
array([-8.08951237e-16, -1.72006384e-17,  1.78678147e-15])
>>> np.std(Z_train, axis=0)[:3]         # ...and have unit variance.
array([1., 1., 1.])
>>> np.mean(Z_test, axis=0)[:3]         # The columns of Z_test each have mean
array([-0.02355067,  0.11665332, -0.03996177])                  # close to 0...
>>> np.std(Z_test, axis=0)[:3]          # ...and have close to unit deviation.
array([0.9263711 , 1.18461151, 0.91548103])

# Check to see if the classification improved.
>>> knn.fit(X_train, y_train).score(X_test, y_test)             # Old score.
0.8951048951048951
>>> knn.fit(Z_train, y_train).score(Z_test, y_test)             # New score.
0.958041958041958
```

This particular transformer is implemented as `sklearn.preprocessing.StandardScaler`. A close cousin is `sklearn.preprocessing.RobustScaler`, which ignores outliers when choosing the scaling and shifting factors.

Like estimators, transformers may have both hyperparameters (provided to the constructor) and model parameters (determined by `fit()`). Thus a transformer looks and acts like an estimator, with the exception of the `predict()` and `transform()` methods.

Achtung!

The `transform()` method should only rely on model parameters derived from the training data in `fit()`, **not** on the data that is worked on in `transform()`. For example, if the `NormalizingTransformer` is fit with the input $\widehat{X}$, then `transform()` should shift and scale any input X by the mean and standard deviation of $\widehat{X}$, not by the mean and standard deviation of X. Otherwise, the transformation is different for each input X.

Scikit-learn Module	**Classifier** Name	Notable Hyperparameters
discriminant_analysis	LinearDiscriminantAnalysis	solver, shrinkage, n_components
discriminant_analysis	QuadraticDiscriminantAnalysis	reg_param
ensemble	AdaBoostClassifier	n_estimators, learning_rate
ensemble	RandomForestClassifier	n_estimators, max_depth
linear_model	LogisticRegression	penalty, C
linear_model	SGDClassifier	loss, penalty, alpha
naive_bayes	GaussianNB	priors
naive_bayes	MultinomialNB	alpha
neighbors	KNeighborsClassifier	n_neighbors, weights
neighbors	RadiusNeighborsClassifier	radius, weights
neural_network	MLPClassifier	hidden_layer_size, activation
svm	SVC	C, kernel
tree	DecisionTreeClassifier	max_depth
Scikit-learn Module	**Regressor** Name	Notable Hyperparameters
ensemble	AdaBoostRegressor	n_estimators, learning_rate
ensemble	ExtraTreesRegressor	n_estimators, max_depth
ensemble	GradientBoostingRegressor	n_estimators, max_depth
ensemble	RandomForestRegressor	n_estimators, max_depth
isotonic	IsotonicRegression	y_min, y_max
kernel_ridge	KernelRidge	alpha, kernel
linear_model	LinearRegression	fit_intercept
neural_network	MLPRegressor	hidden_layer_size, activation
svm	SVR	C, kernel
tree	DecisionTreeRegressor	max_depth
Module	**Transformer** Name	Notable Hyperparameters
decomposition	PCA	n_components
preprocessing	Imputer	missing_values, strategy
preprocessing	MinMaxScaler	feature_range
preprocessing	OneHotEncoder	categorical_features
preprocessing	QuantileTransformer	n_quantiles, output_distribution
preprocessing	RobustScaler	with_centering, with_scaling
preprocessing	StandardScaler	with_mean, with_std

Table D.2: Common scikit-learn classifiers, regressors, and transformers. For full documentation on these classes, see `http://scikit-learn.org/stable/modules/classes.html`.

Validation Tools

Knowing how to determine whether or not an estimator performs well is an essential part of machine learning. This often turns out to be a surprisingly sophisticated issue that largely depends on the type of problem being solved and the kind of data that is available for training. Scikit-learn has validation tools for many situations; for brevity, we restrict our attention to the simple (but important) case of *binary classification*, where the range of the desired model is $Y = \{0, 1\}$.

Evaluation Metrics

The `score()` method of a scikit-learn estimator representing the model $f : X \to \{0, 1\}$ returns the *accuracy* of the model, which is the percent of labels that are predicted correctly. However, accuracy isn't always the best measure of success. Consider the *confusion matrix* for a classifier, the matrix where the (i, j)th entry is the number of observations with actual label i but that are classified as label j. In binary classification, calling the class with label 0 the *negatives* and the class with label 1 the *positives*, this becomes the following.

$$\begin{array}{r} \\ \text{Actual: } 0 \\ \text{Actual: } 1 \end{array} \begin{array}{c} \begin{array}{cc} \text{Predicted: } 0 & \text{Predicted: } 1 \end{array} \\ \begin{bmatrix} \text{True Negatives } (TN) & \text{False Positives } (FP) \\ \text{False Negatives } (FN) & \text{True Positives } (TP) \end{bmatrix} \end{array}$$

With this terminology, we define the following metrics.

- *Accuracy*: $\dfrac{TN + TP}{TN + FN + FP + TP}$, the percent of labels predicted correctly.
- *Precision*: $\dfrac{TP}{TP + FP}$, the percent of predicted positives that are actually correct.
- *Recall*: $\dfrac{TP}{TP + FN}$, the percent of actual positives that are predicted correctly.

Precision is useful in situations where false positives are dangerous or costly, while recall is important when avoiding false negatives takes priority. For example, an email spam filter should avoid filtering out an email that isn't actually spam, so precision is a valuable metric for the filter. On the other hand, recall is more important in disease detection: it is better to test positive and not have the disease than to test negative when the disease is actually present. Focusing on a single metric often leads to skewed results (for example, always predicting the same label), so the following metric is also common.

- *F_β Score*: $(1 + \beta^2)\dfrac{\text{precision} \cdot \text{recall}}{(\beta^2 \cdot \text{precision}) + \text{recall}} = \dfrac{(1 + \beta^2)TP}{(1 + \beta^2)TP + FP + \beta^2 FN}$.

Choosing $\beta < 1$ weighs precision more than recall, while $\beta > 1$ prioritizes recall over precision. The choice of $\beta = 1$ yields the common F_1 score, which weighs precision and recall equally. This is an important alternative to accuracy when, for example, the training set is heavily unbalanced with respect to the class labels.

Scikit-learn implements these metrics in `sklearn.metrics`, as well as functions for evaluating regression, non-binary classification, and clustering models. The general syntax for such functions is `some_score(actual_labels, predicted_labels)`. For the complete list and further discussion, see `http://scikit-learn.org/stable/modules/model_evaluation.html`.

```
>>> from sklearn.metrics import (confusion_matrix, classification_report,
...                              accuracy_score, precision_score,
...                              recall_score, f1_score)

# Fit the esimator to training data and predict the test labels.
>>> knn.fit(X_train, y_train)
>>> knn_predicted = knn.predict(X_test)

# Compute the confusion matrix by comparing actual labels to predicted labels.
>>> CM = confusion_matrix(y_test, knn_predicted)
>>> CM
array([[44,  5],
       [10, 84]])

# Get accuracy (the "usual" score), precision, recall, and f1 scores.
>>> accuracy_score(y_test, knn_predicted)   # (CM[0,0] + CM[1,1]) / CM.sum()
0.8951048951048951
>>> precision_score(y_test, knn_predicted)  # CM[1,1] / CM[:,1].sum()
0.9438202247191011
>>> recall_score(y_test, knn_predicted)     # CM[1,1] / CM[1,:].sum()
0.8936170212765957
>>> f1_score(y_test, knn_predicted)
0.9180327868852459

# Get all of these scores at once with classification_report().
>>> print(classification_report(y_test, knn_predicted))
              precision    recall  f1-score   support

           0       0.81      0.90      0.85        49
           1       0.94      0.89      0.92        94

   micro avg       0.90      0.90      0.90       143
   macro avg       0.88      0.90      0.89       143
weighted avg       0.90      0.90      0.90       143
```

Cross Validation

The `sklearn.model_selection` module has utilities to streamline and improve model evaluation.

- `train_test_split()` randomly splits data into training and testing sets (we already used this).
- `cross_val_score()` randomly splits the data and trains and scores the model a set number of times. Each trial uses different training data and results in a different model. The function returns the score of each trial.
- `cross_validate()` does the same thing as `cross_val_score()`, but it also reports the time it took to fit, the time it took to score, and the scores for the test set as well as the training set.

Doing multiple evaluations with different testing and training sets is extremely important. If the scores on a cross validation test vary wildly, the model is likely overfitting to the training data.

```
>>> from sklearn.model_selection import cross_val_score, cross_validate

# Make (but do not train) a classifier to test.
>>> knn = KNeighborsClassifier(n_neighbors=3)

# Test the classifier on the training data 4 times.
>>> cross_val_score(knn, X_train, y_train, cv=4)
array([0.88811189, 0.92957746, 0.96478873, 0.92253521])

# Get more details on the train/test procedure.
>>> cross_validate(knn, X_train, y_train, cv=4,
...                return_train_score=False)
{'fit_time': array([0.00064683, 0.00042295, 0.00040913, 0.00040436]),
 'score_time': array([0.00115728, 0.00109601, 0.00105286, 0.00102782]),
 'test_score': array([0.88811189, 0.92957746, 0.96478873, 0.92253521])}

# Do the scoring with an alternative metric.
>>> cross_val_score(knn, X_train, y_train, scoring="f1", cv=4)
array([0.93048128, 0.95652174, 0.96629213, 0.93103448])
```

> **Note**
>
> Any estimator, even a user-defined class, can be evaluated with the scikit-learn tools presented in this section as long as that class conforms to the scikit-learn API discussed previously (i.e., inheriting from the correct base classes, having `fit()` and `predict()` methods, managing hyperparameters and parameters correctly, and so on). Any time you define a custom estimator, following the scikit-learn API gives you instant access to tools such as `cross_val_score()`.

Grid Search

Recall that the *hyperparameters* of a machine learning model are user-provided parameters that do not depend on the training data. Finding the optimal hyperparameters for a given model is a challenging and active area of research.[2] However, brute-force searching over a small hyperparameter space is simple in scikit-learn: a `sklearn.model_selection.GridSearchCV` object is initialized with an estimator, a dictionary of hyperparameters, and cross validation parameters (such as `cv` and `scoring`). When its `fit()` method is called, it does a cross validation test on the given estimator with every possible hyperparameter combination.

For example, a k-neighbors classifier has a few important hyperparameters that can have a significant impact on the speed and accuracy of the model: `n_neighbors`, the number of nearest neighbors allowed to vote; and `weights`, which specifies a strategy for weighting the distances between points. The following code tests various combinations of these hyperparameters.

[2] Intelligent hyperparameter selection is sometimes called *metalearning*. See, for example, [SGCP+18].

```
>>> from sklearn.model_selection import GridSearchCV

>>> knn = KNeighborsClassifier()
# Specify the hyperparameters to vary and the possible values they should take.
>>> param_grid = {"n_neighbors": [2, 3, 4, 5, 6],
...               "weights": ["uniform", "distance"]}
>>> knn_gs = GridSearchCV(knn, param_grid, cv=4, scoring="f1", verbose=1)
>>> knn_gs.fit(X_train, y_train)
Fitting 4 folds for each of 5 candidates, totalling 20 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent worker.
[Parallel(n_jobs=1)]: Done  20 out of  20 | elapsed:    0.1s finished

# After fitting, the gridsearch object has data about the results.
>>> print(knn_gs.best_params_, knn_gs.best_score_)
{'n_neighbors': 5, 'weights': 'uniform'} 0.9532526583188765
```

The cost of a grid search rapidly increases as the hyperparameter space grows. However, the outcomes of each trial are completely independent of each other, so the problem of training each classifier is *embarassingly parallel*. To parallelize the grid search over n cores, set the `n_jobs` parameter to n, or set it to -1 to divide the labor between as many cores as are available.

In some circumstances, the parameter grid can be also organized in a way that eliminates redundancy. Consider an `SVC` classifier from `sklearn.svm`, an estimator that works by lifting the data into a high-dimensional space, then constructing a hyperplane to separate the classes. The `SVC` has a hyperparameter, `kernel`, that determines how the lifting into higher dimensions is done, and for each choice of kernel there are additional corresponding hyperparameters. To search the total hyperparameter space without redundancies, enter the parameter grid as a list of dictionaries, each of which defines a different section of the hyperparameter space. In the following code, doing so reduces the number of trials from $3 \times 2 \times 3 \times 4 = 72$ to only $1 + (1 \times 1 \times 3) + (1 \times 4) = 11$.

```
>>> from sklearn.svm import SVC

>>> svc = SVC(C=0.01, max_iter=100)
>>> param_grid = [
...     {"kernel": ["linear"]},
...     {"kernel": ["poly"], "degree": [2,3], "coef0": [0,1,5]},
...     {"kernel": ["rbf"], "gamma": [.01, .1, 1, 100]}]
>>> svc_gs = GridSearchCV(svc, param_grid,
                          cv=4, scoring="f1",
                          verbose=1, n_jobs=-1).fit(X_train, y_train)
Fitting 4 folds for each of 11 candidates, totalling 44 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  44 out of  44 | elapsed:    2.4s finished

>>> print(svc_gs.best_params_, svc_gs.best_score_)
{'gamma': 0.01, 'kernel': 'rbf'} 0.8909310239174055
```

See `https://scikit-learn.org/stable/modules/grid_search.html` for more details about `GridSearchCV` and its relatives.

Pipelines

Most machine learning problems require at least a little data preprocessing before estimation in order to get good results. A scikit-learn *pipeline* (`sklearn.pipeline.Pipeline`) chains together one or more transformers and one estimator into a single object, complete with `fit()` and `predict()` methods. For example, it is often a good idea to shift and scale data before feeding it into a classifier. The `StandardScaler` transformer can be combined with a classifier with a pipeline. Calling `fit()` on the resulting object calls `fit_transform()` on each successive transformer, then `fit()` on the estimator at the end. Likewise, calling `predict()` on the `Pipeline` object calls `transform()` on each transformer, then `predict()` on the estimator.

```
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.pipeline import Pipeline

# Chain together a scaler transformer and a KNN estimator.
>>> pipe = Pipeline([("scaler", StandardScaler()),      # "scaler" is a label.
                     ("knn", KNeighborsClassifier())])  # "knn" is a label.
>>> pipe.fit(X_train, y_train)
>>> pipe.score(X_test, y_test)
0.972027972027972                       # Already an improvement!
```

Since `Pipeline` objects behaves like estimators (following the `fit()` and `predict()` conventions), they can be used with tools like `cross_val_score()` and `GridSearchCV`. To specify which hyperparameters belong to which steps of the pipeline, precede each hyperparameter name with `<stepname>__`. For example, `knn__n_neighbors` corresponds to the `n_neighbors` hyperparameter of the part of the pipeline that is labeled `knn`.

```
# Specify the possible hyperparameters for each step.
>>> pipe_param_grid = {"scaler__with_mean": [True, False],
...                    "scaler__with_std": [True, False],
...                    "knn__n_neighbors": [2,3,4,5,6],
...                    "knn__weights": ["uniform", "distance"]}

# Pass the Pipeline object to the GridSearchCV and fit it to the data.
>>> pipe = Pipeline([("scaler", StandardScaler()),
                     ("knn", KNeighborsClassifier())])
>>> pipe_gs = GridSearchCV(pipe, pipe_param_grid,
...                        cv=4, n_jobs=-1, verbose=1).fit(X_train, y_train)
Fitting 4 folds for each of 40 candidates, totalling 160 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done 160 out of 160 | elapsed:    0.3s finished

>>> print(pipe_gs.best_params_, pipe_gs.best_score_, sep='\n')
{'knn__n_neighbors': 6, 'knn__weights': 'distance',
 'scaler__with_mean': True, 'scaler__with_std': True}
0.971830985915493
```

Pipelines can also be used to compare different transformations or estimators. For example, a pipeline could end in either a `KNeighborsClassier()` or an `SVC()`, even though they have different hyperparameters. Like before, use a list of dictionaries to specify the hyperparameter space.

```
>>> pipe = Pipeline([("scaler", StandardScaler()),
                     ("classifier", KNeighborsClassifier())])
>>> pipe_param_grid = [
...     {"classifier": [KNeighborsClassifier()],    # Try a KNN classifier...
...      "classifier__n_neighbors": [2,3,4,5],
...      "classifier__weights": ["uniform", "distance"]},
...     {"classifier": [SVC(kernel="rbf")],         # ...and an SVM classifier.
...      "classifier__C": [.001, .01, .1, 1, 10, 100],
...      "classifier__gamma": [.001, .01, .1, 1, 10, 100]}]
>>> pipe_gs = GridSearchCV(pipe, pipe_param_grid,
...                        cv=5, scoring="f1",
...                        verbose = 1, n_jobs=-1).fit(X_train, y_train)
Fitting 5 folds for each of 44 candidates, totalling 220 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done 220 out of 220 | elapsed:    0.6s finished

>>> params = pipe_gs.best_params_
>>> print("Best classifier:", params["classifier"])
Best classifier: SVC(C=10, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma=0.01, kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)

# Check the best classifier against the test data.
>>> confusion_matrix(y_test, pipe_gs.predict(X_test))
array([[48,  1],                        # Near perfect!
       [ 1, 93]])
```

Additional Material

Exercises

Problem 1. Writing custom scikit-learn transformers is a convenient way to organize the data cleaning process. Consider the data in `titanic.csv`, which contains information about passengers on the maiden voyage of the *RMS Titanic* in 1912. Write a custom transformer class to clean this data, implementing the `transform()` method as follows:

1. Extract a copy of data frame with just the `"Pclass"`, `"Sex"`, and `"Age"` columns.
2. Replace `NaN` values in the `"Age"` column (of the copied data frame) with the mean age. The mean age of the training data should be calculated in `fit()` and used in `transform()` (compare this step to using `sklearn.preprocessing.Imputer`).
3. Convert the `"Pclass"` column datatype to pandas categoricals (`pd.CategoricalIndex`).
4. Use `pd.get_dummies()` to convert the categorical columns to multiple binary columns (compare this step to using `sklearn.preprocessing.OneHotEncoder`).
5. Cast the result as a NumPy array and return it.

Ensure that your transformer matches scikit-learn conventions (it inherits from the correct base classes, `fit()` returns `self`, etc.).

Problem 2. Read the data from `titanic.csv` with `pd.read_csv()`. The `"Survived"` column indicates which passengers survived, so the entries of the column are the labels that we would like to predict. Drop any rows in the raw data that have `NaN` values in the `"Survived"` column, then separate the column from the rest of the data. Split the data and labels into training and testing sets. Use the training data to fit a transformer from Problem 1, then use that transformer to clean the training set, then the testing set. Finally, train a `LogisticRegressionClassifier` and a `RandomForestClassifier` on the cleaned training data, and score them using the cleaned test set.

Problem 3. Use `classification_report()` to score your classifiers from Problem 2. Next, do a grid search for each classifier (using only the cleaned training data), varying at least two hyperparameters for each kind of model. Use `classification_report()` to score the resulting best estimators with the cleaned test data. Try changing the hyperparameter spaces or scoring metrics so that each grid search yields a better estimator.

Problem 4. Make a pipeline with at least two transformers to further process the Titanic dataset. Do a gridsearch on the pipeline and report the hyperparameters of the best estimator.

Bibliography

[ADH+01] David Ascher, Paul F Dubois, Konrad Hinsen, Jim Hugunin, Travis Oliphant, et al. Numerical python, 2001.

[BLB+13] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[Hun07] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[Oli06] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[Oli07] Travis E Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3), 2007.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[SGCP+18] Brandon Schoenfeld, Christophe Giraud-Carrier, Mason Poggemann, Jarom Christensen, and Kevin Seppi. Preprocessor selection for machine learning pipelines. *arXiv preprint arXiv:1810.09942*, 2018.

[VD10] Guido VanRossum and Fred L Drake. *The python language reference*. Python software foundation Amsterdam, Netherlands, 2010.