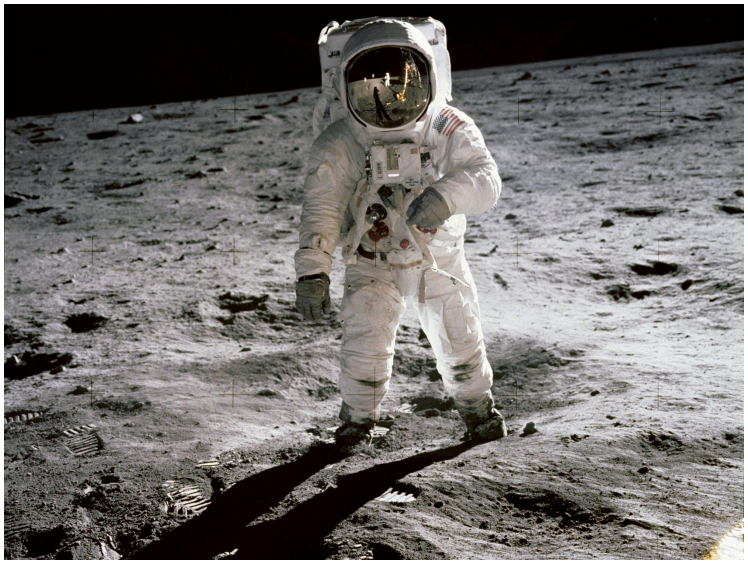
Labs for Foundations of Applied Mathematics

Data Science Essentials

Jeffrey Humpherys & Tyler J. Jarvis, managing editors



List of Contributors

B. Barker
Brigham Young University

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

K. Baldwin
Brigham Young University

J. Bejarano
Brigham Young University

J. Bennett
Brigham Young University

A. Berry
Brigham Young University

Z. Boyd
Brigham Young University

M. Brown
Brigham Young University

A. Carr
Brigham Young University

C. Carter
Brigham Young University

S. Carter
Brigham Young University

T. Christensen
Brigham Young University

M. Cook
Brigham Young University

M. Cutler
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

T. Gledhill
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

S. Halverson
Brigham Young University

E. Hannesson
Brigham Young University

K. Harmer
Brigham Young University

J. Henderson
Brigham Young University

J. Hendricks
Brigham Young University

A. Henriksen
Brigham Young University

I. Henriksen
Brigham Young University

B. Hepner
Brigham Young University

C. Hettinger
Brigham Young University

S. Horst
Brigham Young University

R. Howell
Brigham Young University

E. Ibarra-Campos
Brigham Young University

K. Jacobson
Brigham Young University

R. Jenkins
Brigham Young University

J. Larsen
Brigham Young University

J. Leete
Brigham Young University

Q. Leishman
Brigham Young University

J. Lytle
Brigham Young University

E. Manner
Brigham Young University

M. Matsushita
Brigham Young University

R. McMurray
Brigham Young University

S. McQuarrie
Brigham Young University

E. Mercer
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

C. Noorda
Brigham Young University

A. Oldroyd
Brigham Young University

A. Oveson
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

N. Sill
Brigham Young University

D. Smith
Brigham Young University

J. Smith
Brigham Young University

P. Smith
Brigham Young University

M. Stauffer
Brigham Young University

E. Steadman
Brigham Young University

J. Stewart
Brigham Young University

S. Suggs
Brigham Young University

A. Tate
Brigham Young University

T. Thompson
Brigham Young University

B. Trendler
Brigham Young University

M. Victors
Brigham Young University

E. Walker
Brigham Young University

J. Webb
Brigham Young University

R. Webb
Brigham Young University

J. West
Brigham Young University

R. Wonnacott
Brigham Young University

A. Zaitzeff
Brigham Young University

Preface

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

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.

You can picture the file system as an tree, with directories as branches. Smaller branches stem from bigger branches, and all bigger branches eventually stem from the root of the tree. Similarly, in the Unix file system we have a "root directory", where all other directories are nested in. We denote it by using a single slash (/). All absolute paths originate at the root directory, which means all absolute path strings begin with the / character.

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.

```
~$ pwd
/home/username
```

To see the all the contents of your current directory, type the command `ls` to "list" the files.

```
~$ 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, within the section for Problem 1, to do the following:

- Change into the `Shell1/` directory.
- Print a string (without using `echo`) telling you the 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 check that each directory was deleted, created, or changed correctly.

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 files within Documents/ containing "Mary"
# -r tells grep to search all files within 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 occurrences of `word` within `filename`
`grep -nr "word" dir1`	Find all occurrences 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 syntax 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 if you are confident in the authenticity of the host.

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. The computer with host name `acme20.byu.edu` contains a file that is 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` to this computer, your username is your Net ID, and your password is your typical Net ID password. To use `scp` or `ssh` to access this computer, you will have to be on campus using BYU Wifi (or eduroam), or you can install a BYU specific VPN as explained at `https://it.byu.edu/it?id=kb_article&sys_id=32da18c2877ad110df3c635d0ebb3551`.

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.)

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`. Sometimes after logging onto the machine with `ssh`, there will appear to not be any directories you can access. Using the command `cd /` will fix this. When logging on initially, you also may get a message about not having a `myacmeshare`; this is not needed for this lab and the message may be ignored safely.

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
```

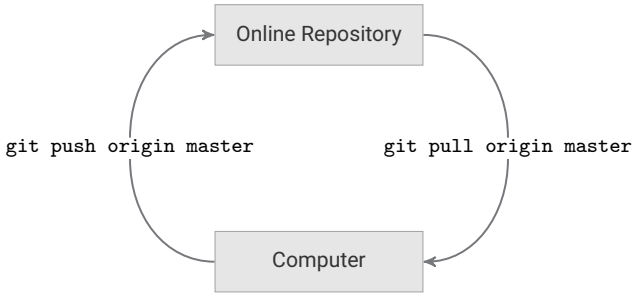


Figure 1.1: Exchanging git commits between the repository and a local clone.

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://github.com/<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` for this example (`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 Unix lab, the majority of learning will not be had in finishing the problems, but in following the examples.*

Archiving and Compression

In file management, the terms archiving and compressing are commonly used interchangeably. However, these are quite different. Archiving is combining a certain number of files into one file. The resulting file will be the same size as the group of files that were archived. Compressing takes a file or group of files and shrinks the file size as much as possible. The resulting compressed file will need to be extracted before being used.

The ZIP file format is common for archiving and compressing files. If the zip Unix command is not installed on your system, you can download it by running

```
>>> sudo apt-get install zip
```

Note that you will need to have administrative rights to download this package. To unzip a file, use `unzip`.

> **Note**
>
> To begin this lab, unzip the `Shell2.zip` file into your `UnixShell2/` directory using a terminal command.

```
# Unzip a zipped file using the unzip command.
$ unzip Shell2.zip
Archive Shell2.zip
   creating: Shell2/
   creating: Shell2/Test/
  inflating: Shell2/.DS_Store
   creating: Shell2/Scripts/
```

```
extracting: Shell2/Scripts/fiteen_secs
extracting: Shell2/Scripts/script3
extracting: Shell2/Scripts/hello.sh...
```

While the `zip` file format is more popular on the Windows platform, the `tar` utility is more common in the Unix environment.

> **Note**
>
> When submitting this lab, you will need to archive and compress your entire `Shell2/` directory into a file called `Shell2.tar.gz` and push `Shell2.tar.gz` as well as `shell2.py` to your online repository.
>
> If you are doing multiple submissions, make sure to delete your previous `Shell2.tar.gz` file before creating a new one from your modified `Shell2/` directory. Refer to `Unix1` for more information on deleting files.
>
> As a final note, please do not push the entire directory to your online repository. *Only* push `Shell2.tar.gz` and `shell2.py`.

The example below demonstrates how to archive and compress our `Shell2/` directory. The `-z` flag calls for the `gzip` compression tool, the `-v` flag calls for a verbose output, the `-p` flag tells the tool to preserve file permission, and the `-f` flag indicates the next parameter will be the name of the archive file. Note that the `-f` flag must always come last.

```
# Remove your archive tar.gz file if you already have one.
$ rm -v Shell2.tar.gz
removed 'Shell2.tar.gz'

# Create a new one from your update Shell2 directory content.
# Remember that the * is the wildcard that represents all strings.
$ tar -zcpf Shell2.tar.gz Shell2/*
```

Working with Files

Displaying File Contents

The unix file system presents many opportunities for the manipulation, viewing, and editing of files. Before moving on to more complex commands, we will look at some of the commands available to view the content of a file.

The `cat` command, followed by the filename, will display all the contents of a file on the terminal screen. This can be problematic if you are dealing with a large file. There are a few available commands to control the output of `cat` in the terminal. See Table 2.1.

As an example, use `less <filename>` to restrict the number of lines that are shown. With this command, use the arrow keys to navigate up and down and press `q` to exit.

Command	Description
`cat`	Print all of the file contents
`more`	Print the file contents one page at a time, navigating forwards
`less`	Like more, but you navigate forward and backwards
`head`	Print the first 10 lines of a file
`head -n`k	Print the first k lines of a file
`tail`	Print the last 10 lines of a file
`tail -n`k	Print the last k lines of a file

Table 2.1: Commands for printing file contents

Pipes and redirects

To combine terminal commands, we can use *pipes*. When we combine or *pipe* commands, the output of one command is passed to the other. We pipe commands together using the `|` (*bar*) operator. In the example directly below, the `cat` command output is piped to `wc -l` (`wc` stands for word count, and the `-l` flag tells the `wc` command to count lines).

In the second part of the example, `ls -s` is piped to `sort -nr`. Refer to the *Unix 1* lab for explanations of `ls` and `sort`. Recall that the `man` command followed by an additional command will output details on the additional command's possible flags and what they mean (for example `man sort`).

```
$ cd Shell2/Files/Feb
# Output the number of lines in assignments.txt.
$ cat assignments.txt | wc -l
9
# Sort the files by file size and output file names and their size.
$ls -s | sort -nr
4 project3.py
4 project2.py
4 assignments.txt
4 pics
total 16
```

In addition to *piping* commands together, when working with files specifically, we can use `redirects`. A *redirect*, represented as `<` in the terminal, passes the file to a terminal command.

To save a command's output to a file, we can use `>` or `>>`. The `>` operator will overwrite anything that may exist in the output file whereas `>>` will append the output to the end of the output file. Examples of *redirects* and writing to a file are given below.

```
# Gets the same result as the first command in the above example.
$ wc -l < assignments.txt
9
# Writes the number of lines in the assignments.txt file to word_count.txt.
$ wc -l < assignments.txt >> word_count.txt
```

Problem 1. The `words.txt` file in the `Documents/` directory contains a list of words that are not in alphabetical order. Write an alphabetically sorted list of words in `words.txt` to a new file in your `Documents/` called `sortedwords.txt` using pipes and redirects. After you write the alphabetized words to the designated file, also write the number of words in `words.txt` to the end of `sortedwords.txt`. Save this file in the `Documents/` directory. Try to accomplish this with a total of two commands or fewer.

Resource Management

To be able to optimize performance, it is valuable to be aware of the resources, specifically hard drive space and computer memory, being used.

Job Control

One way to monitor and optimize performance is in job control. Any time you start a program in the terminal (you could be running a script, opening ipython, etc.,) that program is called a `job`. You can run a job in the foreground and also in the background. When we run a program in the foreground, we see and interact with it. Running a script in the foreground means that we will not be able to enter any other commands in the terminal while the script is running. However, if we choose to run it in the background, we can enter other commands and continue interacting with other programs while the script runs.

Consider the scenario where we have multiple scripts that we want to run. If we know that these scripts will take awhile, we can run them all in the background while we are working on something else. Table 2.2 lists some common commands that are used in job control. We strongly encourage you to experiment with some of 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 2.2: Job control commands

The `fifteen_secs` and `five_secs` scripts in the `Scripts/` directory take fifteen seconds and five seconds to execute respectively. The python file `fifteen_secs.py` in the `Python/` directory takes fifteen seconds to execute, this file counts to fifteen and then outputs *"Success!"*. These will be particularly useful as you are experimenting with these commands.

Remember, that when you use the ./ command in place of other commands you will probably need to change permissions. For more information on changing permissions, review *Unix 1*. Run the following command sequence from the Shell2 directory.

```
# Remember to add executing permissions to the user.
$ ./Scripts/fifteen_secs &
$ python Python/fifteen_secs.py &
$ jobs
[1]+  Running       ./Scripts/fifteen_secs &
[2]-  Running       python Python/fifteen_secs.py &
$ kill %1
[1]-  Terminated    ./Scripts/fifteen_secs &
$ jobs
[1]+  Running       python Python/fifteen_secs.py &
# After the python script finishes it outputs the results.
$ Success!
# To move on, click enter after "Success!" appears in the terminal.

# List all current processes
$ ps
  PID TTY          TIME CMD
    6 tty1     00:00:00 bash
   44 tty1     00:00:00 ps

$ ./Scripts/fifteen_secs &
$ ps
  PID TTY          TIME CMD
    6 tty1     00:00:00 bash
   59 tty1     00:00:00 fifteen_secs
   60 tty1     00:00:00 sleep
   61 tty1     00:00:00 ps

# Stop fifteen_secs
$ kill 59
$ ps
  PID TTY          TIME CMD
    6 tty1     00:00:00 bash
   60 tty1     00:00:00 sleep
   61 tty1     00:00:00 ps
[1]+  Terminated              ./fifteen_secs
```

Problem 2. In addition to the `five_secs` and `fifteen_secs` scripts, the `Scripts/` folder contains three scripts (named `script1`, `script2`, and `script3`) that each take about forty-five seconds to execute. From the `Scripts` directory, execute each of these commands in the background in the following order; `script1`, `script2`, and `script3`. Do this 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.
(Hint: In order to get the same output as the solutions file, you need to run the ./ command and not the *bash* command.)

Using Python for File Management

OS and Glob

Bash 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 2.3 relates some of the common shell commands to Python functions, most of which come from the `os` module in the standard library.

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 2.3: Shell-Python compatibility

In addition to these, Python has a few extra functions that are useful for file management and shell commands. See Table 2.4. The two functions `os.walk()` and `glob.glob()` are especially useful for doing searches like `find` and `grep`. Look at the example below and then try out a few things on your own to try to get a feel for them.

Function	Description
`os.walk()`	Iterate through the subfolders and subfolder files 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 2.4: Other useful Python functions for shell operations.

```
# Your output may differ from the example's output.
>>> 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/fifteen_secs.py
 'Python/mult.py',
```

```
 'Python/project.py']

# Get the names of all .jpg files in any subdirectory.
# The recursive parameter lets '**' match more than one directory.
>> glob("**/*.jpg", recursive=True)
['Photos/IMG_1501.jpg',
 'Photos/img_1879.jpg',
 'Photos/IMG_2164.jpg',
 'Photos/IMG_2379.jpg',
 'Photos/IMG_2182.jpg',
 'Photos/IMG_1510.jpg',
 'Photos/IMG_2746.jpg',
 'Photos/IMG_2679.jpg',
 'Photos/IMG_1595.jpg',
 'Photos/IMG_2044.jpg',
 'Photos/img_1796.jpg',
 'Photos/IMG_2464.jpg',
 'Photos/img_1987.jpg',
 'Photos/img_1842.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/hello.sh
./Scripts/organize_photos.sh
```

Problem 3. Write a Python function `grep()` that accepts the name of a target string and a file pattern. Find all files in the current directory or its subdirectories that match the file pattern. Next, check within the contents of the matched file for the target string. For example, `grep("range", "*.py")` should search Python files for the command `range`. Return a list of the file paths that matched the file pattern *and* the target string. For example, if you're in the Shell2/ directory and your grep function matches the '`calc.py`' file then your grep should return '`Python/calc.py`'

The Subprocess module

The `subprocess` module allows Python to execute actual shell commands in the current working directory. Some important commands for executing shell commands from the `subprocess` module are listed in Table 2.5.

```
$ cd Shell2/Scripts
$ python
```

Function	Description
`subprocess.call()`	run a Unix command
`subprocess.check_output()`	run a Unix command and record its output
`subprocess.check_output.decode()`	this translates Unix command output to a string
`subprocess.Popen()`	use this to pipe togethether Unix commands

Table 2.5: Python subprocess module important commands

```
>>> import subprocess
>>> subprocess.call(["ls", "-l"])
total 40
-rw-r--r--  1 username  groupname  20 Aug 26  2016 five_secs
-rw-r--r--  1 username  groupname  21 Aug 26  2016 script1
-rw-r--r--  1 username  groupname  21 Aug 26  2016 script2
-rw-r--r--  1 username  groupname  21 Aug 26  2016 script3
-rw-r--r--  1 username  groupname  21 Aug 26  2016 fiften_secs
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',
 '-rw-r--r--  1 username  groupname  21 Aug 26  2016 script1',
 '-rw-r--r--  1 username  groupname  21 Aug 26  2016 script2',
 '-rw-r--r--  1 username  groupname  21 Aug 26  2016 script3',
 '-rw-r--r--  1 username  groupname  21 Aug 26  2016 fiften_secs',
 '']
```

`Popen` is a class of the subprocess module, with its own atrributes and commands. It pipes together a few commands, similar to we did at the beginning of the lab. This allows for more versatility in the shell input commands. If you wish to know more about the `Popen` class, go to the subprocess documentation on the internet.

```
$ cd Shell2
$ python
>>> import subprocess
>>> args = ["cat Files/Feb/assignments.txt | wc -l"]
# shell = True indicates to open a new shell process
# note that task is now an object of the Popen class
>>> task = subprocess.Popen(args, shell=True)
>>> 9
```

ACHTUNG!

If shell commands depend on user input, the program is vulnerable to a *shell injection attack*. This applies to Unix Shell commands as well as other situations like web browser interaction with web servers. Be extremely careful when creating a shell process from Python. There are specific functions, like shlex.quote(), that quote specific strings that are used to construct shell commands. But, when possible, it is often better to avoid user input altogether. 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 by force deleting everything in the root directory.[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 4. Using os.path and Glob, write a Python function that accepts an integer n. Search the current directory and all subdirectories for the n largest files. Then sort the list of filenames from the largest to the smallest files. Next, write the line count of the smallest file to a file called `smallest.txt` into the current directory. Finally, return the list of filenames, including the file path, in order from largest to smallest.
(Hint: the shell commands `ls -s` shows the file size.)

As a note, same as in problem 3, to get this problem correct, you need to return the entire file path **starting from the directory that was searched and continuing to the name of the file**. Do not return just the filenames, or the complete file path. For example, if you are currently in the UnixShell2/ directory, meaning the next directory down to be searched will be Shell2, then Shell2/ should be the first part in the names of largest files returned by your function. More concretely, if `'data.txt'` is one files your function will return then instead of returning just `'data.txt'` or all of `'YourComputerSpecificFilePath/UnixShell2/Shell2/Files/Mar/docs/data.txt'` as part of your list, you would return only `'Shell2/Files/Mar/docs/data.txt'`. Notice the paths returned will vary based on both the current working directory you're in and what directories were searched. However, also make sure that your file paths do not begin with './' (Hint: To avoid the additional './' in your file path, use Glob instead of os.walk.)

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. This means that `wget` is capable of following links and directory structure when downloading content.

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
```

Achtung!

If you're using Git Bash for Windows, wget is not automatically included. In order to use it, you have to find a download for wget.exe online. After installing it, you need to include it in the correct directory for git. If git is installed in the default location, you'll need to add it to C:\Program Files\Git\mingw64\bin.

Problem 5. 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.

sed and awk

`sed` and `awk` are two different scripting languages in their own right. `sed` is a stream editor; it performs basic transformations on input text. `Awk` is a text processing language that manipulates and reports data. 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.

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 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 find and replace. We can perform this function on the output of another command, 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.

```
$ cd Shell2/Documents
$ ls -l | awk ' {print $1, $9} '
total
-rw-r--r--. assignments.txt
-rw-r--r--. doc1.txt
-rw-r--r--. doc2.txt
-rw-r--r--. doc3.txt
-rw-r--r--. doc4.txt
-rw-r--r--. files.txt
-rw-r--r--. lines.txt
-rw-r--r--. newfiles.txt
-rw-r--r--. people.txt
-rw-r--r--. review.txt
-rw-r--r--. urls.txt
-rw-r--r--. words.txt
```

Notice we pipe the output of `ls -l` to `awk`. When calling a command using `awk`, we have to use quotation marks. 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.

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. This is done using the `%` command within the `printf` command, which allows us to edit how the relevant data is printed. Look at the last part of the example below to see how it is done.

```
# 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 space at the beginning of run using BEGIN
# 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 different ↩
    order
$ 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 justified, then two characters right justified, then print the last field to its full length.

Problem 6. Inside the `Documents/` directory, you should find a file named `files.txt`. This file contains details on approximately one hundred files. The different fields in the file are separated by tabs. Using `awk`, `sort`, pipes, and redirects, write it to a new file in the current directory named `date_modified.txt` with the following specifications:

- in the first column, print the date the file was modified
- in the second column, print the name of the file
- sort the file from newest to oldest based on the date last modified

All of this can be accomplished using one command.
(Hint: change the field separator to account for tab-delimited files by setting `FS = "\t"` in the `BEGIN` command)

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`.

NOTE

Remember to archive and compress your `Shell2` directory before pushing it to your online repository for grading.

Additional Material

Customizing the Shell

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 with `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.

System Management

In this section, we will address some of the basics of system management. As an introduction, the commands in Table 2.6 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 2.6: Commands for system administration.

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.

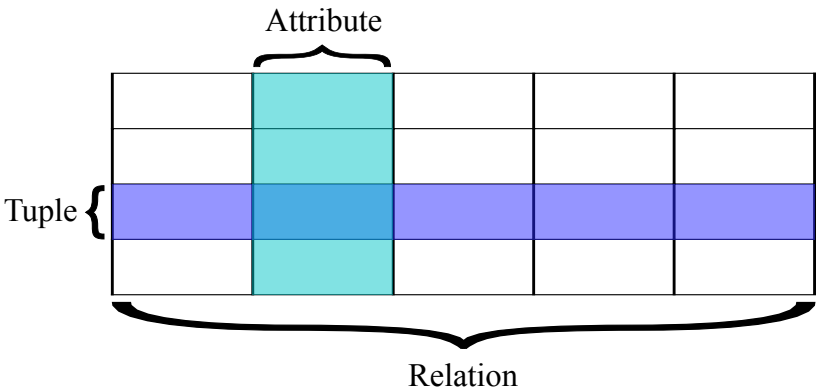


Figure 3.1: See `https://en.wikipedia.org/wiki/Relational_database`.

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, which we use in this lab.

Achtung!

The goal of this lab is to introduce the language SQL, not to teach a specific implementation of SQL. In this lab, we will use the `sqlite3` library, which allows us to interact with a *SQLite* database file. *SQLite* is lightweight, and does not require additional setup, making it great for small projects like this lab and databases which will only be used by a single program at once. For example, Android applications frequently use *SQLite* databases for local storage. However, it is not suitable for many data science applications, since it is difficult to scale across computers and programs. There are a variety of databases that use SQL-like languages designed for different purposes, including *MySQL*, which scales much better and manages concurrent connections efficiently but requires a server and a significant amount of overhead to run.

Although these databases have minor differences in SQL syntax, the concepts taught in this lab are extremely transferrable across all SQL-like derivatives. Remember that you will need to change the Python library you use based on which database type you are connecting to.

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.
```

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

```
...     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.

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.

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).

[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.

- `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 (we'll leave them as -1 for now). A CSV (**c**omma-**s**eparated **v**alues) 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.

> **Note**
>
> SQLite treats `=` and `==` as equivalent operators. For clarity, in this lab we will always use `==` when comparing two values, such as in a `WHERE` clause. The only time we use `=` is in `SET` statements. Be aware that some flavors of SQL (such as MySQL) have no concept of `==`, and typing it in a query will return an error.

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

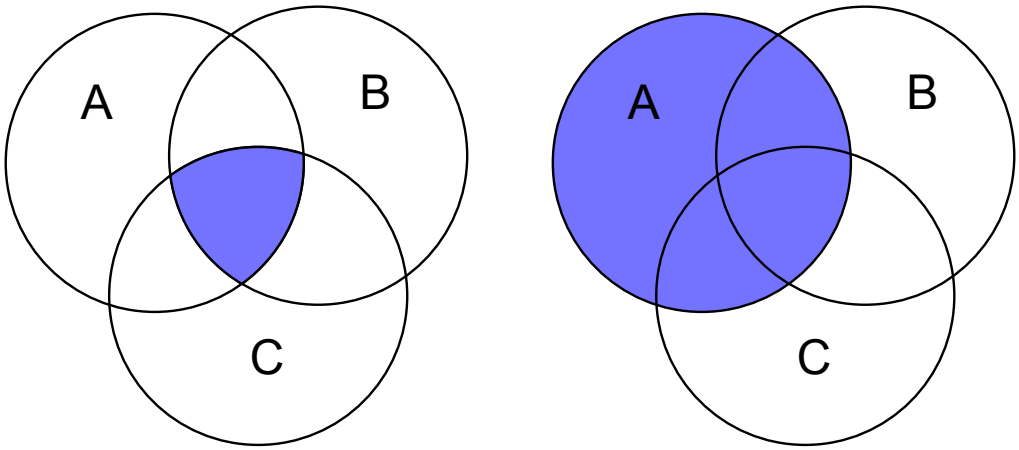
A *join* combines rows from different tables in a database based on common attributes. In other words, a join operation creates a new, temporary table containing data from 2 or more existing tables. Join commands in SQLite have the following general syntax.

```
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+). Be sure to return a list of strings, not a list of tuples of strings.

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 of 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 are then combined by one of the *aggregate functions* `AVG()`, `MIN()`, `MAX()`, `SUM()`, or `COUNT()`. Each of these functions accepts the name of the column to be operated on. Note that the first four of these require the column to hold numerical data. Since our database has no such data, we will delay examples of using the functions other than `COUNT()` until later.

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',)]
```

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 3. 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 descending order by the count `N`, and then in alphabetic order by MajorName. Include **Null** majors.

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']
```

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.

```
# Count how many declared and undeclared majors there are
# The subquery changes NULL values in MajorID to 'Undeclared' and
#    non-NULL to 'Declared'.
>>> 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)]
```

Subqueries can also be joined with other tables, as in the following example. Note also that a subquery can be used to create numerical data out of non-numerical data, which can then be passed into any of the aggregate functions.

```
# Find the proportion of classes each student has an A+, A, or A- in.
# The inner query creates a column 'gradeisa' which is 1 if the student's grade
#    is A+, A, or A-, and 0 otherwise.
>>> cur.execute("SELECT SI.StudentName, AVG(SG.gradeisa) "
...             "FROM ("
...                 "SELECT StudentID, CASE Grade "
...                     "WHEN 'A+' THEN 1 "
...                     "WHEN 'A' THEN 1 "
...                     "WHEN 'A-' THEN 1 "
...                     "ELSE 0 END AS gradeisa "
...                 "FROM StudentGrades) AS SG "
...             "INNER JOIN StudentInfo AS SI "
...             "ON SG.StudentID == SI.StudentID "
...             "GROUP BY SG.StudentID;").fetchall()
[('Cameron Kim', 0.3333333333333333),
 ('Kristopher Tran', 0.5),
 ('Alfonso Phelps', 0.0),
 ('Cassandra Holland', 0.5),
 ('Michelle Fernandez', 0.0),
 ('Roberta Cook', 0.0),
 ('Sammy Burke', 1.0),
 ('Gilbert Chapman', 0.6666666666666666),
 ('Mercedes Hall', 0.3333333333333333),
 ('Rene Cross', 1.0)]
```

Problem 4. 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.

Problem 5. The file `mystery_database.db` contains 4 tables called `table_1`, `table_2`, `table_3`, and `table_4` which contain information on over 5000 subjects. Hidden within these subjects is an obvious outlier. Use what you've learned about SQL to identify the outlier in this database. Return the outlier's name, ID number, eye color, and height as strings in a list. Hint: you may find that joining the tables is more difficult than it's worth; instead, try finding one clue at a time. Most of these subjects lived a long time ago in a galaxy far, far away... so a good place to start might be to find a subject who doesn't meet that criteria. Interesting information about subjects is often included in the `description` column. Also, recall that the following commands can be used to get the column names of a specified table:

```
>>> with sql.connect("database.db") as conn:
...     cur = conn.cursor()
...     cur.execute("SELECT * FROM specified_table;")
...     print([d[0] for d in cur.description])
...
['column_1', 'column_2', 'column_3']
```

The following command might be of use: `cur.execute("PRAGMA case_sensitive_like = TRUE;")`. This command will make the `LIKE` operator case sensitive from the point of execution, until the case sensitivity is turned off.

```
>>> with sql.connect("database.db") as conn:
...     cur = conn.cursor()
...     # Make the LIKE operator hereafter case sensitive
...     cur.execute("PRAGMA case_sensitive_like = TRUE;")
...     cur.execute("SELECT * FROM specified_table "
...                 "WHERE column_1 LIKE 'value';")
```

Achtung!

The goal of this lab is to introduce the language SQL, not to teach a specific implementation of SQL. In this lab, we will use the `sqlite3` library, which allows us to interact with a *SQLite* database file. *SQLite* is lightweight, and does not require additional setup, making it great for small projects like this lab and databases which will only be used by a single program at once. For example, Android applications frequently use *SQLite* databases for local storage. However, it is not suitable for many data science applications, since it is difficult to scale across computers and programs. There are a variety of databases that use SQL-like languages designed for different purposes, including *MySQL*, which scales much better and manages concurrent connections efficiently but requires a server and a significant amount of overhead to run.

Although these databases have minor differences in SQL syntax, the concepts taught in this lab are extremely transferrable across all SQL-like derivatives. Remember that you will need to change the Python library you use based on which database type you are connecting to.

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. This link may be helpful as a reference: https://docs.python.org/3/library/re.html*

A *regular expression* or *regex* is a string of characters that follows a certain syntax to specify a pattern, like generalized shorthand for strings. 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. You can think of the re.compile object as a box with a certain shape cut out of the bottom. When a lot of differently shaped objects are put into the box and shaken around only the objects with the same exact shape as the one cut out of the box will fall out. One method that `re.compile()` uses is the `search()` method, which returns `None` for a string that doesn't match, and a *match* object for a string that does match.

The `match()` method is different than the match object mentioned previously. The match method 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 (analagously, the box you made) 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
```

Assigning re.compile("cat").search("catfish") or re.search("cat","catfish") without the bool() around them to variables will create match objects.

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

Hint: There are online sites like `https://regex101.com/` that can help check answers. Consider building regex expressions one character at a time at this website.

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. Notice that the *pipe* is inside the *group*.

```
>>> 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"
```

Hint: The naive way to do this is create a very long chain of `or` operators with the exact phrases as options. Instead, think about dividing it into two groups of `or` operators where the first group picks the first word and the second groups picks the second word.

There is a file called `test_regular_expressions.py` that contains some prewritten unit tests to help you test your function for this problem.

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 three things: any real number, a single quote followed by any number of non-single-quote characters followed by a single quote (ex: 'example'), 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"`	`"_num = 2.3"`	`"arg_ = 'hey'"`	`"__x__"`	`"var24"`
Non-matches:	`"3rats"`	`"_num = 2.3.2"`	`"arg_ = 'one'two"`	`"sq(x)"`	`" x"`

Matches:	`"max=total"`	`"string= ''"`	`"num_guesses"`
Non-matches:	`"max=2total"`	`"is_4=(value==4)"`	`"pattern = r'^one|two fish$'"`

Hint: It may seem more efficient to keep the equals sign part of the expression outisde of your `or` group (parentheses with pipelines) but that is a really tricky way to do it. It is easier to include the equals sign and space in each case individually. For example, $(= \backslash s^*...| = \backslash s^*...|...)$ instead of $(= \backslash s^*(...|...|...))$.
`Note:` The equals sign is not a special character, but is matching the character '=' exactly. The example above matches an equal sign followed by any number of whitespace.

Unit Test

There is a file called `test_regular_expressions.py` that contains prewritten unit tests for Problem 3. There is a place for you to add your own unit tests for your function in Problem 4 which will be graded.

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 and that any keyword that should have an expression after it does. (Note that this will simplify your regex expression since you won't have to design it to handle cases where some colons are present or to detect the key words that need expressions versus ones that don't.) 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": "01/01/2099",
        "email": "john_doe90@hopefullynotarealaddress.com",
        "phone": "(123)456-7890"
        },
    "Jane Smith": {
        "birthday": None,
        "email": None,
        "phone": "(222)111-3333"
        },
# ...
}
```

Hint: Think about creating a separate re.compile() object to 'catch' each piece of identifying information. Extract and clean each picece individually and build your dictionary incrementally. If the piece of information exists for a specific person, reformat and assign it to that pesron's dictionary. If the information doesn't exist assign it to be `None`.

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 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, BeautifulSoup: Python's canonical tool for efficiently and cleanly navigating and parsing HTML, and how to use Pandas to extract data from HTML tables.*

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. A GET request asks for information from the server, a POST request modifies the state of the server, and a PUT request adds 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`[2], 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/`.

[2] See `https://en.wikipedia.org/wiki/List_of_HTTP_status_codes` for explanation of specific response codes.

```
# 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" />
# ...
```

Attribute	Description
`text`	Content of the response, in unicode
`content`	Content of the response, in bytes
`encoding`	Encoding used to decode response.content
`raw`	Raw socket response, which allows applications to send and obtain packets of information
`status_code`	Numeric code that shows how the action was successful
`ok`	Boolean that is `True` if the `status_code` is less than 400, or `False` if not
`reason`	Text corresponding to the status code

Table 6.1: Different content attributes of the request class.

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. We won't spend much time on that in this lab, but it's 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.

Hint: When writing responses to file you need to use the use the `open()` function with the appropriate file write mode ((either `mode="w"` for plain write mode, or `mode="wb"` for binary write mode). Python interpreters will write the example file the same way whether you use `response.text` or `response.content`.

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]

We will cover this more in the next lab.

[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.

> **Note**
>
> You can open .html files using a text editor or any web browser. In a browser, you can inspect the source code associated with specific elements. Right click the element and select `Inspect`. If you are using Safari, you may first need to enable "Show Develop menu" in "Preferences" under the "Advanced" tab.

BeautifulSoup

BeautifulSoup (`bs4`) is a package[3] 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>
"""
```

[3] BeautifulSoup is not part of the standard library; install it with `pip install beautifulsoup4`.

```
>>> 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>
```

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 6.2: Data attributes of the `bs4.element.Tag` class.

Problem 2. 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.

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 6.3: 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>
```

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. In the next example, we use a tag's `attrs` attribute to access specific attributes within the tag (see Table 6.2).

```
# Get to the <p> tag that has class="story" using these commands.
>>> 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:
...     # Skip the children that are not bs4.element.Tag objects
...     # These don't have the attribute "attrs"
...     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 3. Write a function that reads a file of the same format as the file generated from Problem 1 and load it into BeautifulSoup. Find the first `<a>` tag, and return its text along with a boolean value indicating whether or not it has a hyperlink (`href` attribute).

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 4. 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".

   Hint: the tags that contain the links do not explicitly have "Previous Day" or "Next Day," so you cannot find the tags based on a string. Instead, open the HTML file manually and look at the attributes of the tags that contain the links. Then inspect them to see if there is a better attribute with which to conduct a search.
3. The tag which contains the number associated with the Max Actual Temperature.

Hint: Manually inspect the HTML document in both a browser and in a text editor. Look for a good attribute shared by the Actual Temperatures with which you can conduct a search. Note that the Max Actual Temperature tag will not be the first tag found. Make sure you return the correct tag.

[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: Regular Expressions

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

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

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

```
>>> import re

# Find the first tag with an href attribute containing 'curly'.
>>> pig_soup.find(href=re.compile(r"curly"))
<a class="pig" href="http://example.com/curly" id="link3">Curly.</a

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

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

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

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

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

It is important to note that using Regular Expressions to locate tags will only find tags whose name, attributes, and/or string match exactly. If searching through an HTML file with a Regular expression to find all the tags with an `'id'` attribute with labels that you specify, it will only return the tags found with those labels, not all instances of the labels. Sometimes that is very useful in simplifying Regular Expression searches.

Advanced Search Techniques: CSS Selectors

`BeautifulSoup` also supports the use of CSS selectors. CSS (Cascading Style Sheet) describes the style and layout of a webpage, and CSS selectors provide a useful way to navigate HTML code. Use the method `soup.select()` to find all elements matching an argument. The general format for an argument is `tag-name[attribute-name = 'attribute value']`. The table below lists symbols you can use to more precisely locate various elements.

Symbol	Meaning
`=`	Matches an attribute value exactly
`*=`	Partially matches an attribute value
`^=`	Matches the beginning of an attribute value
`$=`	Matches the end of an attribute value
`+`	Next sibling of matching element
`>`	Search an element's children

Table 6.4: CSS symbols for use with Selenium

You can do many other useful things with CSS selectors. A helpful guide can be found at `https://www.w3schools.com/cssref/css_selectors.asp`. The code below gives an example using arguments described above.

```
# Find all <a> tags with id="link1"
>>> pig_soup.select("[id='link1']")
[<a class="pig" href="http://example.com/larry" id="link1">Larry,</a>]

# Find all tags with an href attribute containing 'curly'.
>>> pig_soup.select("[href*='curly']")
[<a class="pig" href="http://example.com/curly" id="link3">Curly.</a>]

# Find all <a> tags with an href attribute
>>> pig_soup.select("a[href]")
[<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 all <b> tags within a <p> tag with class='title'
 >>> pig_soup.select("p[class='title'] b")
 [<b>The Three Little Pigs</b>]

 # Use a comma to find elements matching one of two arguments
 >>> pig_soup.select("a[href$='mo'],[id='link3']")
[<a class="pig" href="http://example.com/mo" id="link2">Mo</a>,
```

```
<a class="pig" href="http://example.com/curly" id="link3">Curly.</a>]
```

Problem 5. 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 `<a>` 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/`.

Incorporating Pandas with HTML Tables

The Pandas `pd.read_html` method is a neat way to automatically read tables into DataFrames. It works similarly to `BeautifulSoup`, but is streamlined to only scrape all the tables from HTML pages.

```
>>> import pandas as pd
#Read in "file.html"
>>> tables = pd.read_html('file.html')
# Choose the correct table that pd.read_html found as a DataFrame
>>> df = tables[0]
# The DataFrame is now like any other Pandas DataFrame that needs a bit of data↩
    cleaning
>>> df["column_1"]
0       0
1       2
2       .
3       6
4       8
       ..
15     30
16     32
17      .
18      .
19     38
```

Like any other DataFrame table, if there are non-numerical values in a column, the column type will default to strings, so make sure when sorting values by number, set the column data type to numeric with `pd.to_numeric`.

For more help with `pd.read_html` see
`https://pandas.pydata.org/docs/reference/api/pandas.read_html.html`.

Problem 6. The file `large_banks_data.html` is one of the pages from the index in Problem 5.[a] Read the specified file and load it into Pandas.

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.

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

7 Web Crawling

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

Scraping Etiquette

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

1. The scraper doesn't respect the website's terms and conditions or 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.

Recall that consecutive 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()`.

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

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

The amount of necessary wait time depends on the website. Sometimes, `robots.txt` contains a `Crawl-delay` directive which gives a number of seconds to wait between successive requests. 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.

Python provides a parsing library called `urllib.robotparser` for reading `robot.txt` files. Below is an example of using this library to check where robots are allowed on arxiv.org. A website's `robots.txt` file will often include different instructions for specific crawlers. These crawlers are identified by a `User-agent` string. For example, Google's webcrawler, `User-agent` Googlebot, may be directed to index only the pages the website wants to have listed on a Google search. We will use the default `User-agent`, `"*"`.

```
>>> from urllib import robotparser
>>> rp = robotparser.RobotFileParser()
# Set the URL for the robots.txt file. Note that the URL contains `robots.txt'
>>> rp.set_url("https://arxiv.org/robots.txt")
>>> rp.read()
# Request the crawl-delay time for the default User-agent
>>> rp.crawl_delay("*")
15
# Check if User-agent "*" can access the page
>>> rp.can_fetch("*", "https://arxiv.org/archive/math/")
True
>>> rp.can_fetch("*", "https://arxiv.org/IgnoreMe/")
False
```

Problem 1. Write a program that accepts a web address defaulting to the site `http://example.webscraping.com` and a list of pages defaulting to `["/", "/trap", "/places/default/search"]`. For each page, check if the `robots.txt` file permits access. Return a list of boolean values corresponding to each page. Also return the crawl delay time.

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(4):
        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-4" 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 2. Modify `scrape_books()` so that it gathers the price for each fiction book and returns the mean price, in £, of a fiction book.

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 (for example, if you have to scroll down to load more results). 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.

If you are using WSL and Windows, type these this into your Ubuntu terminal:

```
wget dl.google.com/linux/direct/google-chrome-stable_current_amd64.deb
sudo apt install -y ./google-chrome-stable_current_amd64.deb
```

If your program still can't find the driver after you've downloaded it, add the argument `executable_path = "path/to/driver/file"` when you call `webdriver`. If this doesn't work, you may need to add the location to your system PATH.

On a Mac, open the file `/etc/path` and add the new location.

On Linux, add `export PATH="path/to/driver/file:$PATH"` to the file `/.bashrc` .

On non-WSL Windows, follow a tutorial such as this one: `https://www.architectryan.com/2018/03/17/add-to-the-path-on-windows-10/`.

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('css', 'tag_name_here')`	The first tag with the given name
`find_element('name', 'tag_name_here')`	The first tag with the specified `name` attribute
`find_element('class name', 'tag_name_here')`	The first tag with the given `class` attribute
`find_element('id', 'tag_name_here')`	The first tag with the given `id` attribute
`find_element('link text', 'tag_name_here')`	The first tag with a matching `href` attribute
`find_element('partial link text', 'tag_name_here')`	The first tag with a partially matching `href` attribute

Table 7.1: The method of the `selenium.webdriver.Chrome` class.

Note other documentation uses a format like `find_element(By.TAG_NAME, 'tag_name_here')`. To use that format, make sure to use an import statement like `from selenium.webdriver.common.by import By`.

The `find_element()` method 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. If you want to find *all* the matching object, each webdriver also has a `find_elements()` method (element**s**, 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('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 3. The Internet Archive Wayback Machine archives sites so users can find the history of websites they visit and IMDB contains a variety of information on movies. Information on the top 10 box office movies of the weekend of February 21 - 23, 2020 can be found at `https://web.archive.org/web/20200226124504/https://www.imdb.com/chart/boxoffice/?ref_=nv_ch_cht`.

Using BeaufiulSoup, Selenium, or both, return a list, with each title on a new row, 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. Break ties using the weekend gross, from most money to the least.

Using CSS Selectors

In addition to the methods listed in Table 7.1, you can also use CSS or XPath selectors to interact more precisely with the page source. Refer to the CSS symbols for use with Selenium Table from the WebScraping lab or `https://www.w3schools.com/cssref/css_selectors.php` for a review of CSS syntax.

The following code searches Google for "Python Selenium Docs" and then clicks on the second result.

```
#As before, go to Google and type in the search bar,
# but this time we use CSS selectors

>>> from selenium.webdriver.common.keys import Keys
>>> from selenium.common.exceptions import NoSuchElementException
>>> from selenium.webdriver.common.by import By

>>> browser = webdriver.Chrome()
>>> try:
...     browser.get("https://google.com")
...     try:
...         search_bar = browser.find_element(By.CSS_SELECTOR,
...                 "input[name='q']")
...         search_bar.clear()
...         search_bar.send_keys("Python Selenium Docs")
...         search_bar.send_keys(Keys.RETURN)
...         try:
```

```
...             # Wait a second, then get the second search result
...             time.sleep(1)
...             # "+ div" returns the element's  next sibling with a "div" tag
...             second_result = browser.find_element(By.CSS_SELECTOR,
...                     "div[class='g'] + div")
...             try:
...                 # Get the link, which is a child of second_result
...                 link = second_result.find_element(By.CSS_SELECTOR,
...                     "div[class='r']")
...                 link.click()
...                 time.sleep(1)
...             #Remember to handle exceptions
...             except NoSuchElementException:
...                 print("Could not find link")
...         except NoSuchElementException:
...             print("Could not find second result")
...     except NoSuchElementException:
...         print("Could not find the search bar")
... finally:
...     browser.close() #browser.quit() would also work
```

In the above code, we could have used `find_element("class name", "div[class='r']")`, but when you need more precision than that, CSS selectors can be very useful. Remember that to view specific HTML associated with an object in Chrome or Firefox, you can right click on the object and click "Inspect." For Safari, you need to first enable "Show Develop menu" in "Preferences" under "Advanced." Keep in mind that you can also search through the source code (ctrl+f or cmd+f) to make sure you're using a unique identifier.

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, some websites are somewhat defensive about scrapers, but Selenium can sometimes make it possible to gather info without offending the administrators.

Problem 4. *Project Euler* (`https://projecteuler.net`) is a collection of mathematical computing problems. Each problem is listed with an ID, a description/title, and the number of users that have solved the problem.

Using Selenium, BeautifulSoup, or both, record the number of people who have solved each of the 700+ problems in the archive at `https://projecteuler.net/archives`. Plot the number of people who have solved each problem against the problem IDs, using a log scale for the y-axis. Display the scatter plot, then state the IDs of which problems have been solved most and least number of times.

Problem 5. The website `http://watir.com/examples/simple_form.html` contains an example form you can fill out. Using Selenium fill out the form for this example person:

```
First name: Elizabeth
Last name: Bennet
Email: elizabeth.bennet@pemberleymail.com
Interests: Dancing, Books
Want to recieve newsletter: Yes
```

Remember to use a crawl delay of at least 5 seconds so you don't send your requests too fast. Print out the Response text that filling out the form provides.

8 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 and syntax, and we explore the capabilities of pandas for quickly analyzing and presenting data.*

Pandas Basics

Pandas is a python library used primarily to analyze data. It combines functionality of NumPy, MatPlotLib, and SQL to create an 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 an `ndarray`. However, a `Series` has an explicitly defined `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
>>> import numpy as np
# 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 labeled with an `index` (as in a `Series`) and the columns are labeled in the attribute `columns`.

There are many different ways to initialize a `DataFrame`. One way to initialize a `DataFrame` is by 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 corresponding entry for the other `Series` is `NaN`.

We can also initialize a `DataFrame` with a NumPy array. With this method, the data is passed in as a 2-dimensional NumPy array, while the column labels and the index are passed in as parameters. The first column label goes with the first column of the array, the second with the second, and so forth. The index works similarly.

```
# Initialize DataFrame with NumPy array. This is identical to the grades ↩
    DataFrame above.
>>> 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'])

# View the columns
>>> grades.columns
Index(['Math', 'English'], dtype='object')

# View the Index
>>> grades.index
Index(['Barbara', 'David', 'Eleanor', 'Greg', 'Lauren', 'Mark'], dtype='object'↩
    )
```

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.]])
```

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
`read_csv()`	Read a csv and convert into a DataFrame
`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
`read_html()`	Read a table in an html page and convert to a DataFrame

Table 8.1: 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. Some frequently-used keyword arguments include the following:

- `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.

Another particularly useful function is `read_html()`, which is useful when scraping data. It takes in a url or html file and an optional argument `match`, a string or regex, and returns a list of the tables that match the `match` in a DataFrame.

Data Manipulation

Accessing Data

In general, the best way to access data in a `Series` or `DataFrame` is through the indexers `loc` and `iloc`. While array slicing can be used, it is more efficient to use these 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` or `iloc` explicitly bypasses these extra checks. The `loc` index selects rows and columns based on their labels, while `iloc` selects them based on their integer position. With these indexers, the first and second arguments refer to the rows and columns, respectively, just as array slicing.

```
# 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
```

To access an entire column of a `DataFrame`, the most efficient method is to use only square brackets and the name of the column, without the indexer. This syntax can also be used to create a new column or reset the values of an entire column.

```
# Create a 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
```

Datasets can often be very large and thus difficult to visualize. Pandas has various methods to make this easier. 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 entries 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 8.2.

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

Problem 1. The file `budget.csv` contains the budget of a college student over the course of 4 years. Write a function that performs the following operations in this order:

1. Read in `budget.csv` as a `DataFrame` with the index as column 0. Hint: Use `index_col=0` to set the first column as the index when reading in the csv.
2. Reindex the columns such that amount spent on groceries is the first column and all other columns maintain the same ordering.
3. Sort the `DataFrame` in descending order by how much money was spent on `Groceries`.
4. Reset all values in the `'Rent'` column to `800.0`.
5. 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 based off of `ndarray`, most NumPy functions work with pandas structures. For example, basic vector operations work as would be 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` has a variety of other methods similar to NumPy arrays. A collection of these methods is found in Table 8.3.

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 8.3: 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, which can be 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

# 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()` can be used to rank the values in a data set, either within each entry or with each column. This function 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 in their classes in descending order
# (best to worst)
# The method keyword specifies what rank to use when ties occur.
>>> 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 or 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, or we can fill them with some other value, such as the mean. `NaN` might also mean something specific, such as some default value, which should inform what to do with `NaN` values.

```
# 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

# 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 2. Write a function which uses `budget.csv` to answer the questions "Which category affects living expenses the most? Which affects other expenses the most?" Perform the following manipulations:

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

Return the names of each of those columns as a tuple. The first should be of the column corresponding to `'Living Expenses'` and the second to `'Other'`.

Complex Operations in Pandas

Often times, the data that we have is not exactly the data we want to analyze. In cases like this we use more complex data manipulation tools to access only the data that we need.

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 helpful 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'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 3 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   ID          5 non-null      int64
 1   Grade       5 non-null      float64
 2   Math_Major  5 non-null      object
dtypes: float64(1), int64(1), object(1)
memory usage: 248.0+ bytes
```

Masks

Sometimes, we only want to access data from a single column. For example if we want to only access the `ID` of the students in the `studentInfo DataFrame`, then we would use the following syntax.

```
# Get the ID column from studentInfo
>>> studentInfo.ID
0    0
1    1
2    2
3    3
4    4
5    5
6    6
7    7
```

```
8    8
Name: ID, dtype: int64
```

If we want to access multiple columns at once we can use a list of column names.

```
# Get the ID and Age columns.
>>> 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   20
```

Now we can access the specific columns that we want. However, some of these columns may still contain data points that we don't want to consider. In this case we can build a mask. Each mask that we build will return a pandas `Series` object with a `bool` value at each index indicating if the condition is satisfied.

```
# Create a mask for all student receiving financial aid.
>>> mask = otherInfo['Financial_Aid'] == 'y'
# Access other info where the mask is true and display the ID and GPA columns.
>>> otherInfo[mask][['ID', 'GPA']]
   ID  GPA
0   0  3.8
3   3  3.9
7   7  3.4
```

We can also create compound masks with multiple statements. We do this using the same syntax you would use for a compound mask in a normal NumPy array. Useful operators are `&`, the `AND` operator; `|`, the `OR` operator; and $\sim$, the `NOT` operator.

```
# Get all student names where Class = 'J' OR Class = 'Sp'.
>>>  mask = (studentInfo.Class == 'J') | (studentInfo.Class == 'Sp')
>>> studentInfo[mask].Name
0        Mylan
4        Jason
5         Remi
6         Matt
7    Alexander
Name: Name, dtype: object
# This can also be acomplished with the following command:
# studentInfo[studentInfo['Class'].isin(['J','Sp'])]['Name']
```

Problem 3. 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'`. Answer the following questions using the pandas methods learned in this lab. The answer of each question should be saved as indicated. Return the answers to all three questions as a tuple (i.e. `(answer_1,answer_2,answer_3)`).

1. Identify the three crimes that have a mean yearly number of occurences 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. Hint: Consider using `idxmax()` to extract the title.
2. Examine the data from 2000 and later. Sort this data (in ascending order) according to number of murders. Find the years where aggravated assault is greater than 850,000. Save the indices (the years) of the masked and reordered `DataFrame` as a NumPy array to return as the answer.
3. What year had the highest crime rate? In this year, which crime was committed the most? What percentage of the total crime that year was it? Return this percentage as the answer in decimal form (e.g. 50% would be returned as 0.5). Note that the crime rate is #crimes/population.

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 8.4 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 8.4: 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 4. 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 any rows without numerical values, cast the `"VALUE"` column to floats, then return the updated DataFrame.

Hint: You can change the column type the same way you'd change a numpy array type.

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 8.5: 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 8.6 for a sampling of some of the options. For a complete list of the options, see `https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#timeseries-offset-aliases`.

Parameter	Description
`"D"`	calendar daily (default)
`"B"`	business daily (every business day)
`"H"`	hourly
`"T"`	minutely
`"S"`	secondly
`"MS"`	first day of the month (Month Start)
`"BMS"`	first business day of the month (Business Month Start)
`"W-MON"`	every Monday (Week-Monday)
`"WOM-3FRI"`	every 3rd Friday of the month (Week of the Month - 3rd Friday)

Table 8.6: 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 5. The file `paychecks.csv` contains values of an hourly employee's last 93 paychecks. Paychecks are given every other Friday, starting on March 14, 2008, 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`.

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 6. Compute the following information about the DJIA dataset from Problem 4 that has a DateTimeIndex.

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

Return the DateTimeIndex of the day with the largest gain and the day with the largest loss as a tuple.
(Hint: Call your function from Problem 4 to get the DataFrame already cleaned and with DatetimeIndex).

More information on how to use `datetime` with Pandas is in the additional material section. This includes working with `Periods` and more analysis with time series.

Additional Material

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})
```

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 passed 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   20

# SELECT ID, GPA FROM otherInfo WHERE Financial_Aid = 'y'
>>> mask = otherInfo['Financial_Aid'] == 'y'
>>> otherInfo[mask][['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')
   ID   Name Sex  Age Class  Grade Math_Major
0   0  Mylan   M   20    Sp    4.0          y
1   1  Regan   F   21    Se    3.0          n
2   3   Jess   F   22    Se    4.0          n
3   5   Remi   F   20     J    3.5          y
4   6   Matt   M   20     J    3.0          n

# 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
```

More Datetime with Pandas

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 8.6.

```
# 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('2016-01-01 00:00:00')
> p2.end_time
Timestamp('2016-12-31 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]')

# 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]')

# Change frequency to be quarterly.
```

```
>>> q = p.asfreq("Q-DEC")
>>> q
PeriodIndex(['2010Q2', '2010Q3', '2010Q4', '2011Q1'], dtype='period[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
>>> q -= 1
>>> q
PeriodIndex(['2010Q1', '2010Q2', '2010Q3', '2010Q4'], dtype='period[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)
>>> q = q.to_timestamp(how='end')
>>> q
DatetimeIndex(['2010-03-31', '2010-06-30', '2010-09-30', '2010-12-31'],
              dtype='datetime64[ns]', freq='Q-DEC')

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

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 8.6: `"D"` for daily, `"H"` for hourly, and so on.

```
>>> import numpy as np

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

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

```
>>> years.mean()                    # Average entry by year.
2000-12-31    0.491
2001-12-31    0.514
2002-12-31    0.484
                ...
2007-12-31    0.508
2008-12-31    0.521
2009-12-31    0.523
Freq: A-DEC, dtype: float64

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

Elementary Time Series Analysis

Rolling Functions and Exponentially-Weighted Moving Functions

Many time series are inherently noisy. To analyze general trends in data, we use *rolling functions* and *exponentally-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.

```
import matplotlib.pyplot as plt

# Generate a time series using random walk from a uniform distribution.
seed = 42
N = 10000
bias = 0.01

rng = np.random.default_rng(seed)
s = np.zeros(N)
s[1:] = rng.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=0.3, 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.

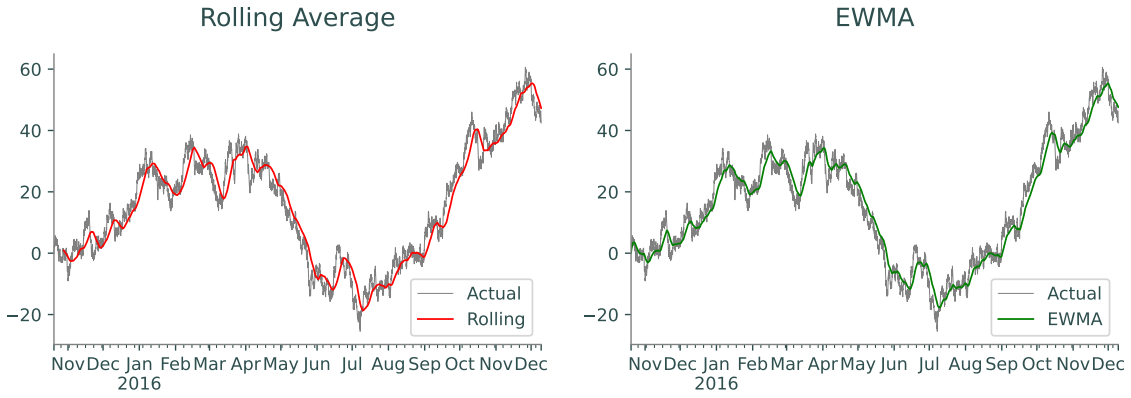
In the case of a *exponentially-weighted moving average* (EWMA), each data point is calculated as follows.

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

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 8.1: Rolling average and EWMA.

```
ax2 = plt.subplot(122)
s.plot(color="gray", lw=0.3, 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")
```

9 Pandas 2: Plotting

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

> **Note**
>
> This lab is designed to be more flexible than other labs. You will be asked to make visualizations customized to your liking. As long as your plots clearly visualize the data, are properly labeled, etc., you will be graded generously.

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.

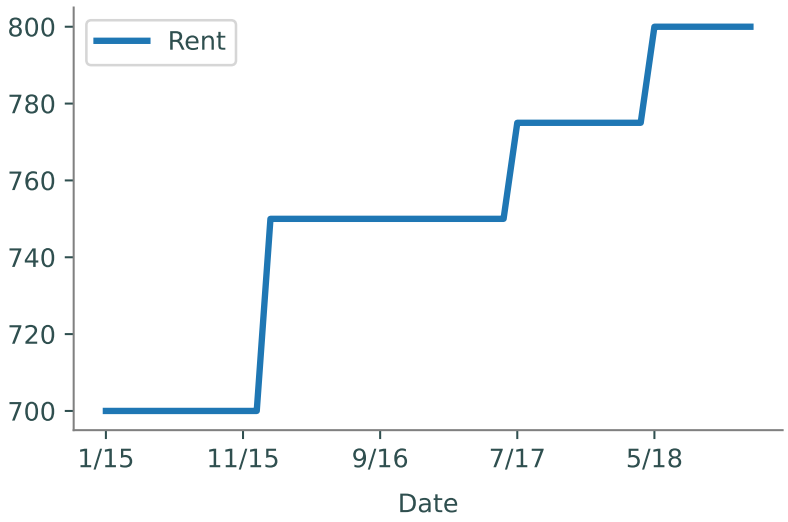
Plot Type	`plot()` ID	Uses and Advantages
Line plot	`"line"`	Show trends ordered in data; easy to compare multiple data sets
Scatter plot	`"scatter"`	Compare exactly two data sets, independent of ordering
Bar plot	`"bar"`, `"barh"`	Compare categorical or sequential data
Histogram	`"hist"`	Show frequencies 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 9.1: 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 functions, 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("budget.csv", index_col="Date")
>>> 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.

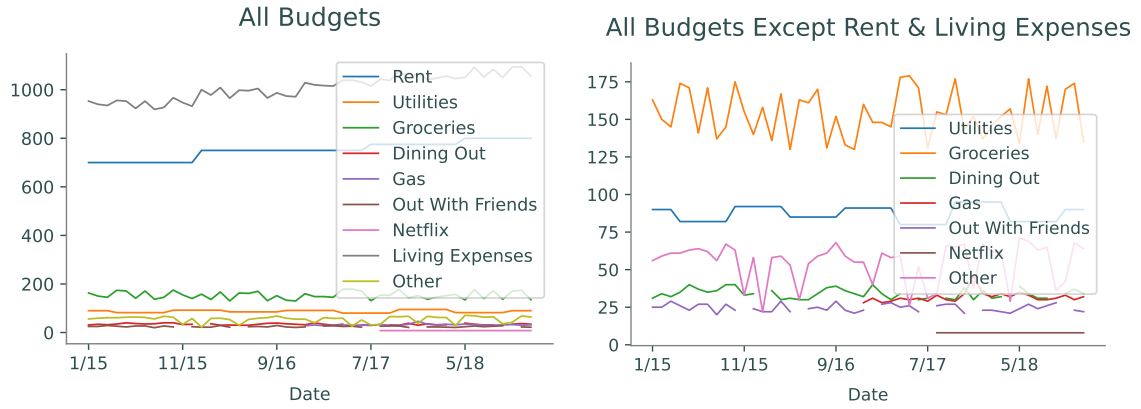
```
>>> plt.plot(budget.index, budget['Rent'], label='Rent')
>>> plt.xlabel(budget.index.name)
>>> plt.xlim(min(budget.index), max(budget.index))
>>> plt.legend(loc='best')
```

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(title="All Budgets", linewidth=1)
>>> budget.drop(["Rent", "Living Expenses"], axis=1).plot(linewidth=1, title="↩
    All Budgets Except Rent & Living Expenses")
```



(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 9.1

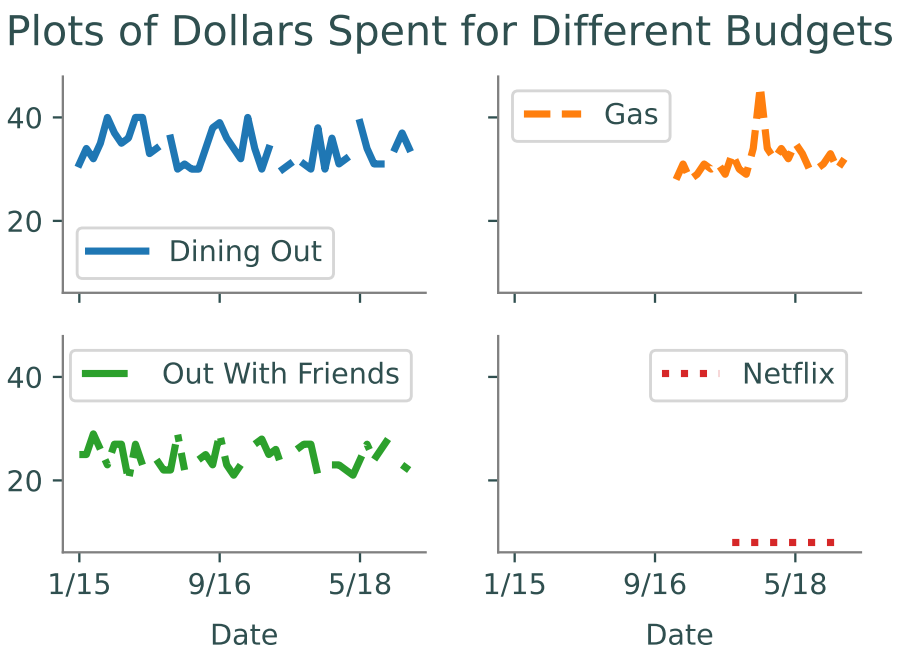
While plotting every `Series` at once can give an overview of all the data, the resulting plot is often difficult for the reader to understand. For example, the budget data set has 9 columns, so the resulting figure, Figure 9.1a, is fairly cluttered.

One way to declutter a visualization is to examine less data. For example, the columns `'Living Expenses'` and `'Rent'` have values that are much larger than the other columns. Dropping these columns gives a better overview of the remaining data, as shown in Figure 9.1b.

> **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=['-', '--', '-.', ':'], title="Plots of Dollars Spent for ↩
    Different Budgets")
```



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=0.7, bins=10, title="Frequency of Amount (in dollars) Spent")

# 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, title="Frequency of Amount (in dollars) Spent")
```

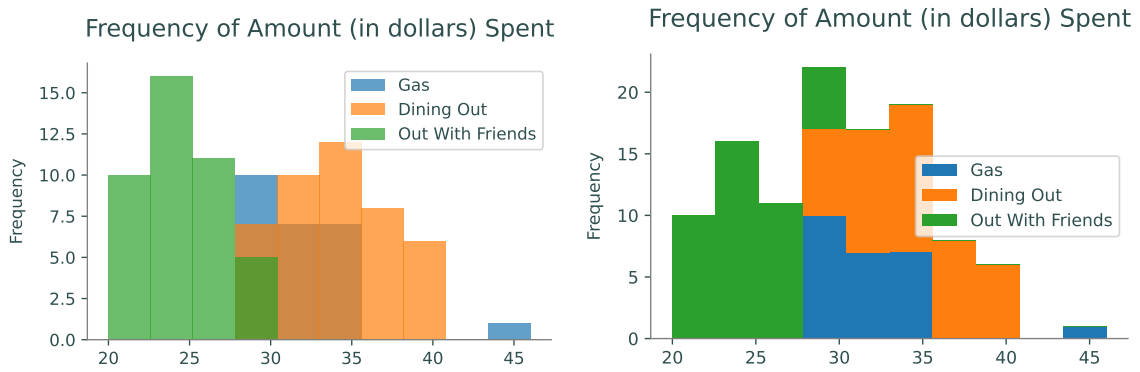


Figure 9.2: Two examples of histograms that are difficult to understand because multiple columns are plotted.

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"]].hist(grid=False, bins=10)
```

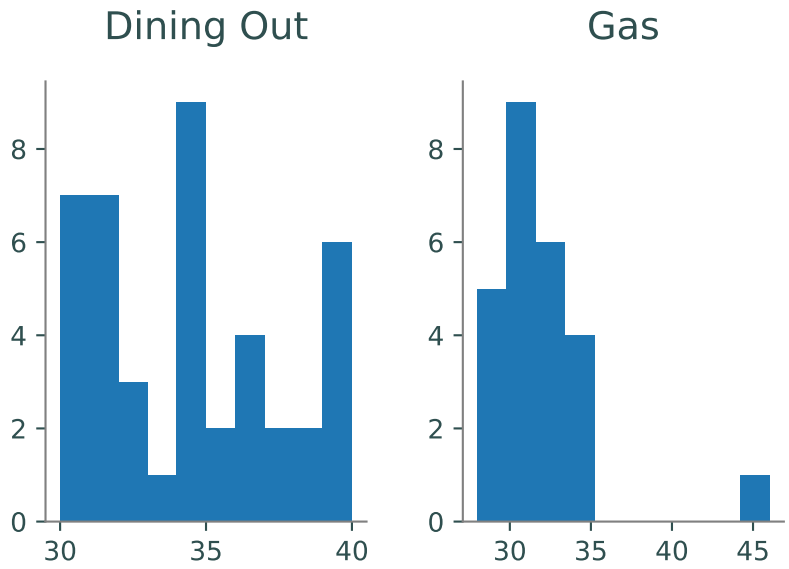


Figure 9.3: Histograms of `"Dining Out"` and `"Gas"`.

Problem 1. Create 3 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.

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 9.1b 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"], title="Amount Spent on ↩
    Dining Out and Out with Friends per Day")

# Make a scatter plot of 'Dining Out' against 'Out With Friends'
>>> budget.plot(kind="scatter", x="Dining Out", y="Out With Friends",
...     alpha=0.8, xlim=(0, max(budget['Dining Out']) + 1),
...     ylim=(0, max(budget['Out With Friends']) + 1),
...     title="Correlation between Dining Out and Out with Friends")
```

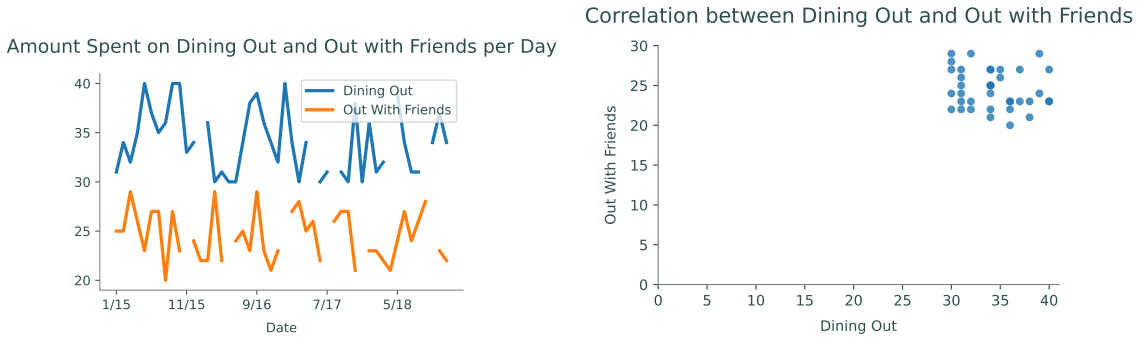


Figure 9.4: Correlations between `"Dining Out"` and `"Out With Friends"`.

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 number of murders is likely accurate across the board, since murder is conspicuous and highly reported. However, the increase of rape incidents is probably caused both by the general rise in crime as well as an increase in the percentage of rape incidents being reported. 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 9.5: 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="barh"`) 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()
```

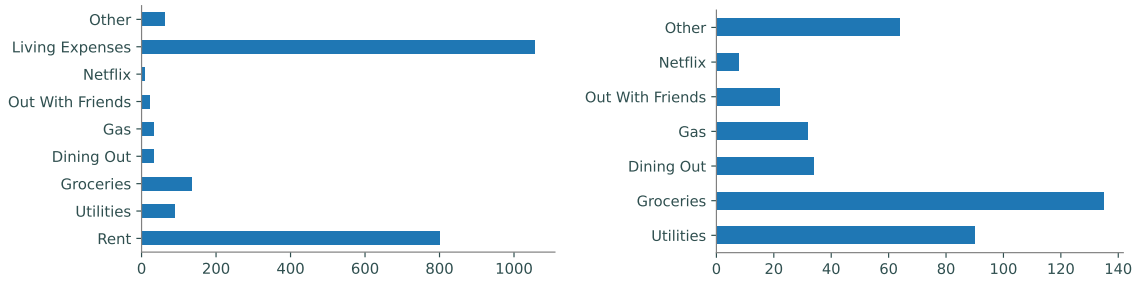


Figure 9.6: Bar graphs showing expenses paid in the last month of `budget`.

Problem 2. Using `crime_data.csv`, plot the trends between `Larceny` and the following variables:

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

Make sure each graph is clearly labeled and readable. One of these variables does *not* have a linear trend with `Larceny`. Return a string identifying this variable (You're welcome to hard-code this value after observing the data).

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)
```

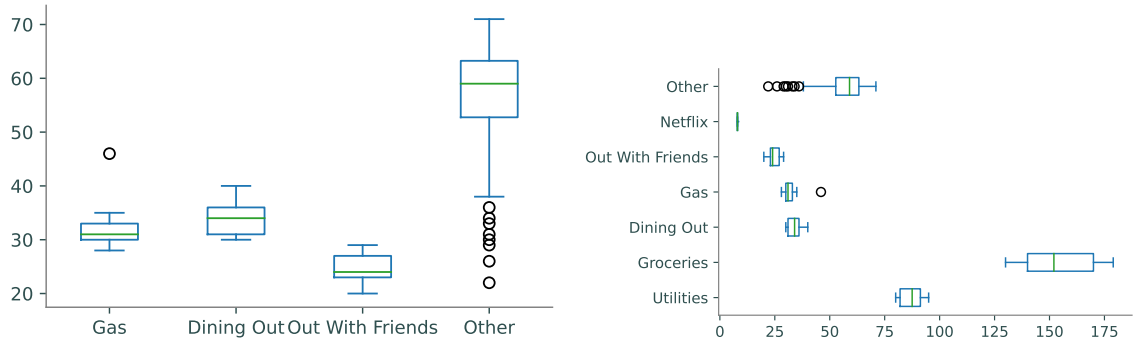


Figure 9.7: Vertical and horizontal box plots of `budget` dataset.

Hexbin Plots

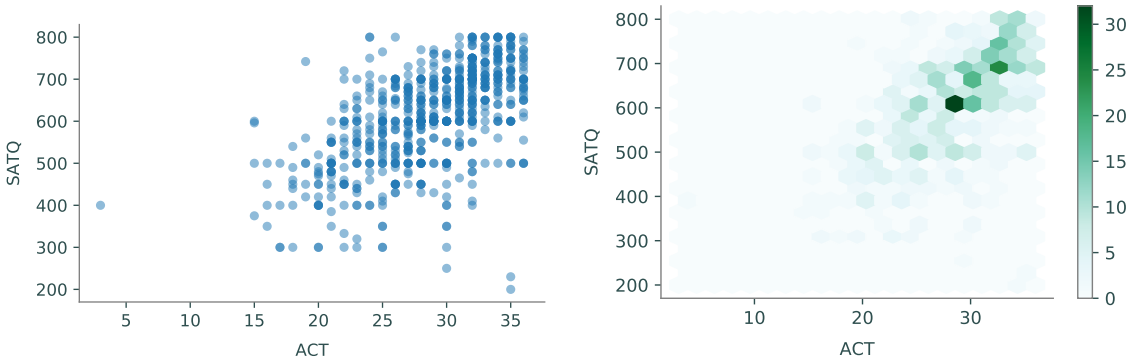
A scatter plot is essentially a plot of samples from the joint distribution of two columns. However, scatter plots can be uninformative for large data sets when the points in a scatter plot are closely clustered. *Hexbin plots* solve this problem by plotting point density in hexagonal bins—essentially 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 9.8a, is highly cluttered, even though the points have some transparency. A hexbin plot of the same data, Figure 9.8b, 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 9.8: 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` as box plots,
2. The distribution of `Vehicle Thefts` against `Robbery` as a hexbin plot.

As usual, all plots should be labeled and easy to read.
Hint: To get the x-axis label to display, you might need to set the `sharex` parameter of `plot()` to `False`.

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 9.9. 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.

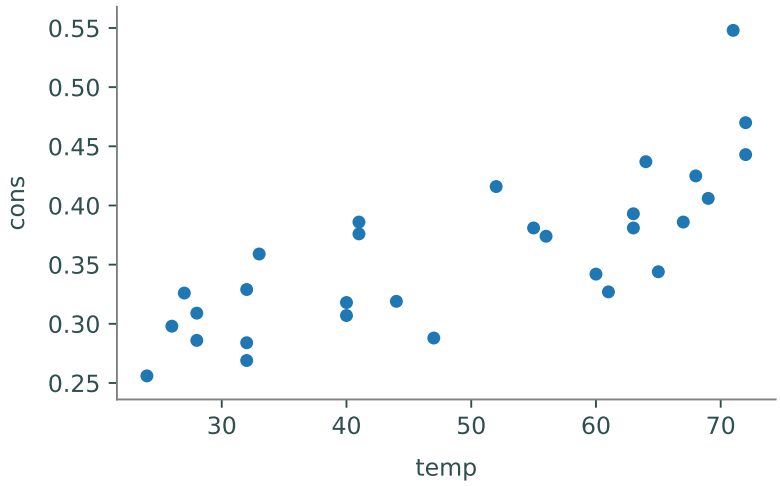


Figure 9.9: Non-specific data.

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 9.9. 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 9.9. 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 9.9 is better understood with the following information:

1. The dataset details ice cream consumption via 30 four-week periods from March 1951 to July 1953 in the United States.
2. `cons` corresponds to "consumption of ice cream per capita" and is measured in pints.
3. `income` is the family's weekly income in dollars.

4. `price` is the price of a pint of ice cream.

5. `temp` corresponds to temperature, degrees Fahrenheit.

6. 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 capita*, 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 labeling using xlabel and ylabel
>>> plt.xlabel("Temp (Fahrenheit)")
>>> plt.ylabel("Consumption per capita (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. The first two parameters of `plt.text` are the x and y coordinates to place the text. The third parameter is the text to write. For instance, using `plt.text(0.5, 0.5, "Hello World")` will center the `Hello World` string in the axes.

```
>>> 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 9.10.

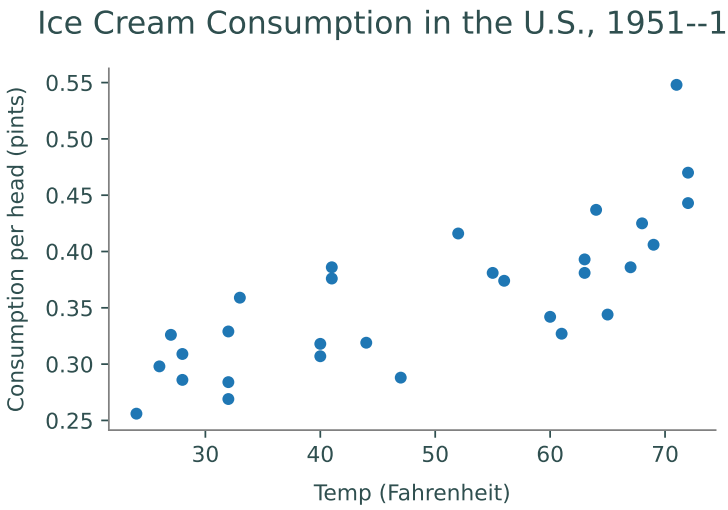


Figure 9.10: Source: Hildreth, C. and J. Lu (1960) *Demand relations with autocorrelated disturbances*, Technical Bulletin No 2765, Michigan State University.

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? Do schools with lower student to faculty ratios have higher tuition costs? etc.). These three plots should be easy to understand and have clear titles and variable names. In addition, the dataset should be cited in your first plot as follows:

James, G., Witten, D., Hastie, T., and Tibshirani, R. (2013) An Introduction to Statistical Learning with applications in R, www.StatLearning.com, Springer-Verlag, New York

This problem is intended to give you flexibility to create your own visualizations that answer your own questions. As long as your plots are clear and include all the information listed above, you will be graded generously!

10 Pandas 3: Grouping

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

Groupby

The file `mammal_sleep.csv`[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")
```

[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"`.

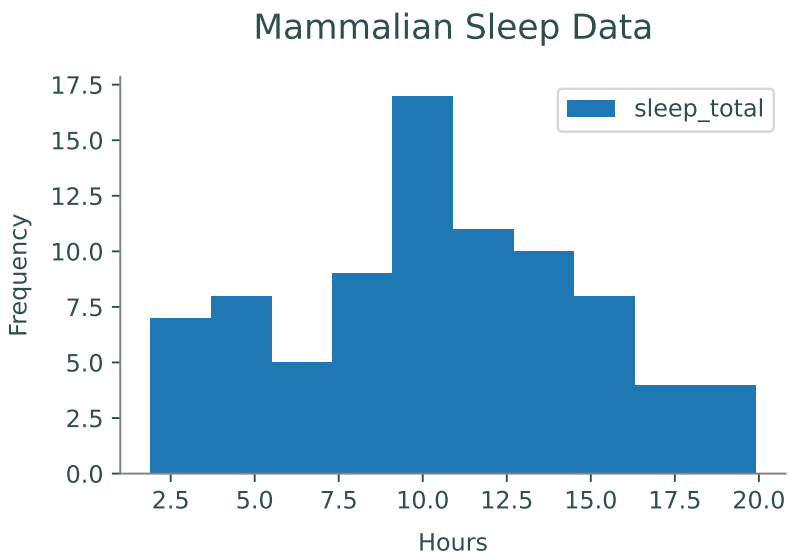


Figure 10.1: `"sleep_total"` frequencies from the mammalian sleep data set.

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

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

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

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

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

# Get a single group and sample a few rows. Note vore='carni' in each entry.
>>> vores.get_group("carni").sample(5)
       name     genus   vore      order  sleep_total  sleep_rem  sleep_cycle
80    Genet   Genetta  carni  Carnivora          6.3        1.3          NaN
50    Tiger  Panthera  carni  Carnivora         15.8        NaN          NaN
8       Dog     Canis  carni  Carnivora         10.1        2.9        0.333
0   Cheetah  Acinonyx  carni  Carnivora         12.1        NaN          NaN
82  Red fox    Vulpes  carni  Carnivora          9.8        2.4        0.350
```

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(numeric_only=True)
         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. Percent of students from the top 10% of their high school class
3. Percent of students from the top 25% 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 we were comparing whether the average number of professors with PhDs was higher at private or public universities, we would find that public universities have a higher average, and we 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 10.1 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")
```

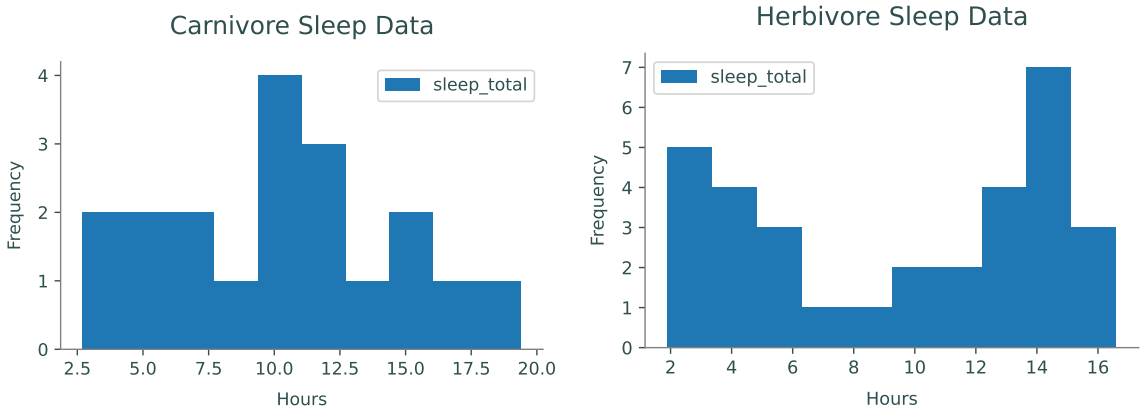
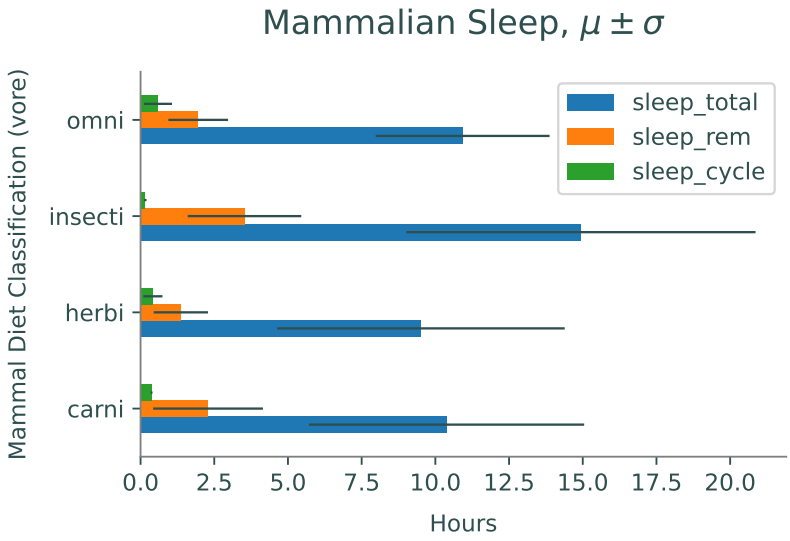


Figure 10.2: `"sleep_total"` histograms for two groups in the mammalian sleep data set.

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

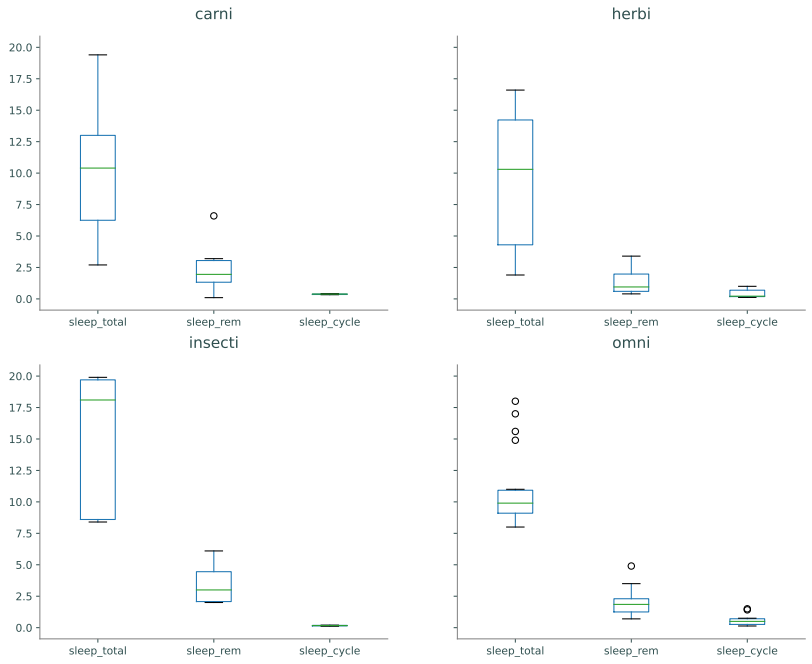
```
>>> cols = ["sleep_total", "sleep_rem", "sleep_cycle"]
>>> vores[cols].mean().plot(kind="barh", xerr=vores[cols].std(),
... title="Mammalian Sleep, $\mu\pm\sigma$")
```

```
>>> plt.xlabel("Hours")
>>> plt.ylabel("Mammal Diet Classification (vore)")
```



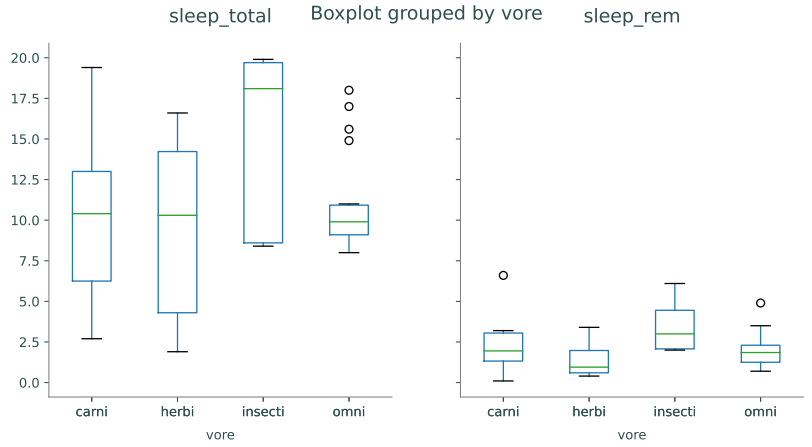
Box plots are well suited for comparing similar distributions. The `boxplot()` method of the `GroupBy` class creates one subplot **per group**, plotting each of the columns as a box plot.

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



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

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



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

Problem 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 does the range of money spent on room and board compare between 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(f"{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 as 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. You can use the keyword argument `aggfunc` to choose among different ways to aggregate the data. For example, if you use `aggfunc='min'`, the value displayed will be the minimum of all the values. Other arguments include `'max'`, `'std'` for standard deviation, `'sum'`, or `'count'` to count the number of occurrences. You also may pass in any function that reduces to a single float, like `np.argmax` or even `np.linalg.norm` if you wish. A list of functions can also be passed into the `aggfunc` keyword argument.

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"] = 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()
```

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

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

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

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

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

# See how many people from each category survived.
>>> titanic.pivot_table(values="Survived", index="Sex", columns="Pclass",
...                     aggfunc="sum")
Pclass    1.0   2.0    3.0
Sex
female  139.0  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. Which race/sex combination has the highest `Usual Weekly Earnings` in total?
2. Which race/sex combination has the lowest cumulative `Usual Hours Worked`?
3. What race/sex combination has the highest average `Usual Hours Worked`?

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. You may hard code each tuple as long as you also provide the working code that gave you the solutions.
Some useful keys in understanding 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])
[(0, 4], (0, 4], (0, 4], (0, 4], (4, 8], (4, 8], (4, 8]]
Categories (2, interval[int64, right]): [(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]   131   85  158
male   (0, 12]      4   11   35
       (12, 18]     4   10   37
       (18, 80]   171  150  421
```

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               131  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 or equal to 43 means the person got at least a bachelor's degree. Out of these categories, which degree type is the most common among the workers in this dataset?

2. Partition the `Age` column into 6 equally-sized groups using `pd.qcut()`. Which interval has the highest average `Usual Hours Worked`?

3. Using the partitions from the first two parts, what age/degree combination has the lowest 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 is 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()`. These can also be obtained from a pivot table, as in the example 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. Partition `perc.alumni` into evenly spaced intervals of 20% using `pd.cut()`. Does the number of both private and public universities decrease as the percentage of alumni that donate increases, as expected?

2. Partition `Grad.Rate` into evenly spaced intervals of 20% using `pd.cut()`. Is the partition with the greatest number of schools the same for private and public universities?

3. Create a column that gives the acceptance rate of each university (using columns `Accept` and `Apps`). Partition this column into evenly spaced intervals of 25% using `pd.cut()`. Is it true that the partition with the least average number of students from the top 10 percent of their high school class is the same for both private and public universities?

4. Use the same partition as part 3. The average percentage of students admitted from the top 10 percent of their high school class is very high in private universities with the lowest acceptance rates ($< 25\%$ acceptance rate). Why is this *not* a good conclusion to draw solely from this dataset? Use only the data to explain why; do not extrapolate.

11 GeoPandas

Lab Objective: *GeoPandas is a package designed to organize and manipulate geographic data. It combines the data manipulation tools of pandas with 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 functionality of pandas with 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 also geometric attributes. GeoPandas can be installed via `pip`:

```
>>> pip install geopandas
```

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 objects such as roads. A polygon could be used to identify regions, such as a country. Multipoints, multilines, and multipolygons contain lists of points, lines, and polygons, respectively.

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 11.1. 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 attribute `bounds` to find the maximum and minimum coordinates of Egypt in the GeoDataFrame `worldmap.gpkg`.

Method/Attribute	Description
`distance(other)`	returns minimum distance from GeoSeries to `other`
`contains(other)`	returns `True` if shape contains `other`
`intersects(other)`	returns `True` if shape intersects `other`
`area`	returns shape area
`convex_hull`	returns convex shape around all points in the object
`bounds`	returns the bounding x- and y-coordinates of the object

Table 11.1: Attributes and Methods for GeoSeries

```
>>> import geopandas as gpd
>>> world = gpd.read_file(worldmap.gpkg)
# Get GeoSeries for Egypt
>>> egypt = world[world['SOVEREIGNT']=='Egypt']

# Find bounds of Egypt
>>> egypt.bounds
         minx    miny         maxx      maxy
163   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`, which must be a GeoSeries. This GeoSeries column is used when a spatial method, like `distance()`, is used on the GeoDataFrame. Therefore all attributes and methods used for GeoSeries can also be used on GeoDataFrame objects.

A GeoDataFrame can be made from a pandas DataFrame. One of the columns in the DataFrame should contain geometric information. That column can be converted to a GeoSeries using the `apply()` method. At this point, the Pandas DataFrame can be cast as a GeoDataFrame. Assign which column will be the `geometry` using either the `geometry` keyword in the constructor or the `set_geometry()` method afterwards.

```
>>> import pandas as pd
>>> import geopandas as gpd
>>> from shapely.geometry import Point, Polygon

# 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 = gpd.GeoDataFrame(df, geometry='Coordinates')

# Equivalently, specify the geometry after construction
# Note that set_geometry() returns a new GeoDataFrame
>>> gdf = gpd.GeoDataFrame(df)
>>> gdf = gdf.set_geometry('Coordinates')

# Display the GeoDataFrame
>>> gdf
           City       Country  Latitude  Longitude                  Coordinates
0         Seoul   South Korea     37.57     126.98   POINT (126.98000 37.57000)
1          Lima          Peru    -12.05     -77.04  POINT (-77.04000 -12.05000)
2  Johannesburg  South Africa    -26.20      28.04   POINT (28.04000 -26.20000)

# Create a polygon with all three cities as points
>>> city_polygon = Polygon(list(zip(df.Longitude, df.Latitude)))
```

A `GeoDataFrame` can also be made directly from a dictionary. If the dictionary already contains geometric objects, the corresponding column can be directly set as the `geometry` in the constructor. Otherwise, a column containing geometry data can be created as in the above example and then set as the `geometry` with the `set_geometry()` method.

```
# Both of these methods create the same GeoDataFrame as above
# Directly create the GeoDataFrame from the dictionary
>>> gdf = gpd.GeoDataFrame({'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 and set as the geometry
>>> gdf['Coordinates'] = list(zip(gdf.Longitude, gdf.Latitude))
>>> gdf['Coordinates'] = gdf['Coordinates'].apply(Point)
# inplace=True modifies gdf itself rather than returning a copy
>>> gdf.set_geometry('Coordinates', inplace=True)

# Equivalently, using a dictionary that already contains geometry objects
>>> gdf = gpd.GeoDataFrame({'City': ['Seoul', 'Lima', 'Johannesburg'],
...                    'Country': ['South Korea', 'Peru', 'South Africa'],
...                    'Coordinates': [Point(126.98,37.57),
...                        Point(-77.04,-12.05), Point(28.04,-12.05)]},
```

```
...                         geometry='Coordinates')
```

Method/Attribute	Description
`abs()`	returns series/dataframe with absolute numeric value of each element
`add(other)`	returns addition of dataframe and `other` element-wise
`affine_transform(matrix)`	returns `GeoSeries` with translated geometries
`append(other)`	returns new object with appended rows of `other` to the end of caller
`dot(other)`	returns dataframe of matrix multiplication with `other`
`equals(other)`	tests if the two objects contain the same elements

Table 11.2: Attributes and Methods for GeoDataFrame

Note

Longitude is the angular measurement starting at the Prime Meridian, 0°, and going to 180° to the east and −180° to the west. 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.

Plotting GeoDataFrames

Information from a GeoDataFrame is plotted based on the geometry column. Data points are displayed as geometry objects. The following example plots the shapes in the `world` GeoDataFrame.

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

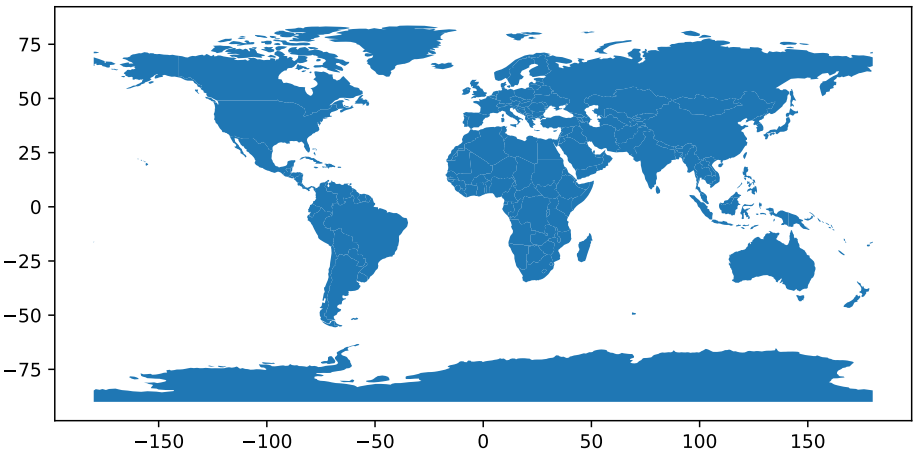


Figure 11.1: World map

Multiple GeoDataFrames can be plotted at once. This can be done by by setting one GeoDataFrame as the base of the plot and ensuring that each layer uses the same axes. In the following example, the file `airports.csv`, containing the coordinates of world airports, is loaded into a GeoDataFrame and plotted on top of the boundary of the `world` GeoDataFrame.

```
# Set outline of world countries as base
>>> fig, ax = plt.subplots(1, figsize=(10,4))
>>> base = world.boundary.plot(edgecolor='black', ax=ax, linewidth=1)

# Load airport data and convert to a GeoDataFrame
>>> airports = pd.read_csv('airports.csv')
>>> airports['Coordinates'] = list(zip(airports.Longitude, airports.Latitude))
>>> airports['Coordinates'] = airports.Coordinates.apply(Point)
>>> airports = gpd.GeoDataFrame(airports, geometry='Coordinates')

# Plot airports on top of world map
>>> airports.plot(ax=base, marker='o', color='green', markersize=1)
>>> ax.set_xlabel('Longitude')
>>> ax.set_ylabel('Latitude')
>>> ax.set_title('World Airports')
```

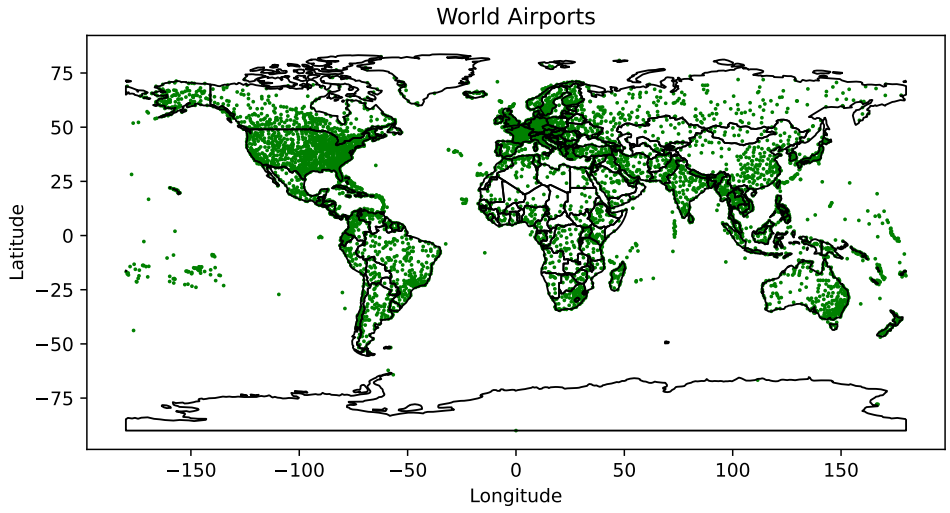


Figure 11.2: Airport map

Problem 1. The file `worldmap.gpkg.zip` contains data of the worldmap used in above examples.[a] After unzipping, use the command `geopandas.read_file('worldmap.gpkg')` to create a GeoDataFrame of this information.

Then, read in the file `airports.csv` as a pandas DataFrame. Create three convex hulls around the three sets of airports listed below. This can be done by passing in lists of the airports' coordinates (Longitude and Latitude zipped together) to a `shapely.geometry.Polygon` object. Finally, create a new GeoDataFrame using a dictionary with key `'geometry'` and with a list of these three Polygons as the value. Plot this GeoDataFrame, and then plot the outlined world map on top of it.

- Maio Airport, Scatsta Airport, Stokmarknes Skagen Airport, Bekily Airport, K. D. Matanzima Airport, RAF Ascension Island
- Oiapoque Airport, Maio Airport, Zhezkazgan Airport, Walton Airport, RAF Ascension Island, Usiminas Airport, Piloto Osvaldo Marques Dias Airport
- Zhezkazgan Airport, Khanty Mansiysk Airport, Novy Urengoy Airport, Kalay Airport, Biju Patnaik Airport, Walton Airport

[a]Source: `https://www.naturalearthdata.com/downloads/110m-cultural-vectors/`

NOTE

`.gpkg` files are actually structured as a directory that contains several files that each contain parts of the data. For instance, `worldmap.gpkg` consists of the files `worldmap.cpg`, `worldmap.dbf`, `worldmap.prj`, `worldmap.shp`, and `worldmap.shx`. Be sure that these files are placed directly in the first level of `worldmap.gpkg`, and not in further subdirectories.

Working with 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 the world GeoDataFrame for GDP_PER_CAPITA
>>> world['GDP_PER_CAP'] = world.GDP_MD / world.POP_EST
```

GeoDataFrames can utilize many pandas functionalities, and 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 objects 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 along, and `aggfunc` tells how to combine the information in all other columns. The default `aggfunc` is `first`, which returns the first application entry. In the following example, we use `sum` as the `aggfunc` so that each continent is the combination of its countries.

```
>>> world = world[['CONTINENT', 'geometry', 'GDP_PER_CAP']]

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

Projections and Coloring

When plotting, GeoPandas uses the CRS (coordinate reference system) of a GeoDataFrame. This reference system indicates how coordinates should be spaced on a plot. Two of the most commonly used CRSs are EPSG:4326 and EPSG:3395. EPSG:4326 is the standard latitude-longitude projection used by GPS. EPSG:3395, also known as the Mercator projection, is the standard navigational projection.

When creating a new GeoDataFrame, it is important to set the `crs` attribute of the GeoDataFrame. This allows any plots to be shown correctly. Furthermore, GeoDataFrames being layered need to have the same CRS. To change the CRS, use the method `to_crs()`.

```
# Check CRS of world GeoDataFrame
>>> print(world.crs)
EPSG:4326

# Change CRS of world to Mercator
# inplace=True ensures that we modify world instead of returning a copy
>>> world.to_crs(3395, inplace=True)
>>> print(world.crs)
EPSG:3395
```

GeoPandas accepts many different CRSs; a reference can be found at `www.spatialreference.org`. Additionally, inspecting a given CRS object in the terminal without using `print()` or `str()` can be used to get additional information about a specific CRS:[1]

```
>>> world.crs
<Projected CRS: EPSG:3395>
Name: WGS 84 / World Mercator
Axis Info [cartesian]:
- E[east]: Easting (metre)
- N[north]: Northing (metre)
Area of Use:
- name: World between 80°S and 84°N.
```

[1] This can also be accomplished using `print(repr(crs))`.

```
- bounds: (-180.0, -80.0, 180.0, 84.0)
Coordinate Operation:
- name: World Mercator
- method: Mercator (variant A)
Datum: World Geodetic System 1984
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

GeoDataFrames can also be plotted using the values in the other attributes of the GeoSeries. 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.

```
>>> fig, ax = plt.subplots(1, figsize=(10,4))
# Plot world based on gdp
>>> world.plot(column='GDP_MD', cmap='OrRd', legend=True,  ax=ax)
>>> ax.set_title('World Map based on GDP')
>>> ax.set_xlabel('Longitude')
>>> ax.set_ylabel('Latitude')
>>> plt.show()
```

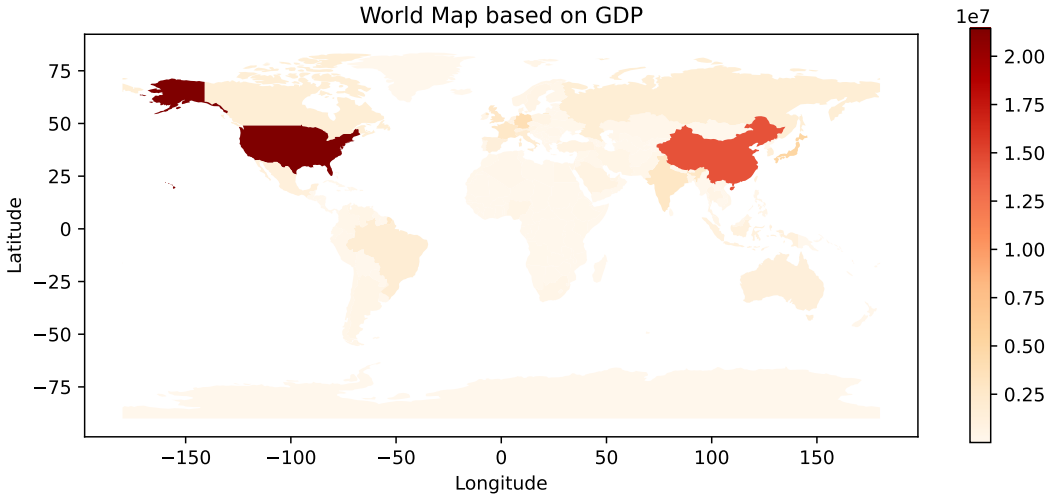


Figure 11.3: World Map Based on GDP

Problem 2. The file `county_data.gpkg.zip` contains information about US counties.[a] After unzipping, use the command `geopandas.read_file('county_data.gpkg')` to create a GeoDataFrame of this information. Each county's shape is stored in the `geometry` column. Use this to plot the boundaries of all US counties two times, first using the default CRS and then using EPSG:5071.

Next, create a new GeoDataFrame that combines (`dissolve`) all counties within each state (`by='STATEFP'`). Drop regions with the following STATEFP codes: 02, 15, 60, 66, 69, 72, 78. Plot the boundary of this GeoDataFrame to see an outline of all 48 contiguous states. Ensure a CRS of EPSG:5071.

[a]Source: `http://www2.census.gov/geo/tiger/GENZ2016/shp/cb_2016_us_county_5m.zip`

Merging GeoDataFrames

Just as multiple pandas DataFrames can be merged, multiple GeoDataFrames can be merged with attribute joins or spatial joins. An attribute join is similar to 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 = gpd.read_file(worldmap.gpkg)
>>> cities = gpd.read_file(cities.gpkg)

# Create subsets of the world and cities GeoDataFrames
>>> world = world[['CONTINENT', 'SOVEREIGNT', 'SOV_A3']]
>>> cities = cities[['NAME', 'SOV_A3']]

# Merge the GeoDataFrames on their SOV_A3 code
>>> countries = world.merge(cities, on='SOV_A3')
```

A spatial join merges two GeoDataFrames based on their geometry data. The function used for this is `sjoin`. `sjoin` accepts two GeoDataFrames and then direction on how to merge. It is imperative that two GeoDataFrames have the same CRS. In the example below, we merge using an `inner` join with the option `intersects`. The `inner` join means that we will only use keys in the intersection of both geometry columns, and we will retain only the left geometry column. `intersects` tells the GeoDataFrames to merge on GeoSeries that intersect each other. Other options include `contains` and `within`.

```
# Combine countries and cities on their geographic location
>>> countries = gpd.sjoin(world, cities, how='inner', predicate='intersects')
```

Problem 3. Load in the file `nytimes.csv`[a] as a Pandas DataFrame. This file includes county-level data for the cumulative cases and deaths of Covid-19 in the US, starting with the first case in Snohomish County, Washington, on January 21, 2020.

Merge the county GeoDataFrame from `county_data.gpkg` with the `nytimes` DataFrame on the county `fips` codes (a FIPS code is a 5-digit unique identifier for geographic locations). Note that the `fips` column of the `nytimes` DataFrame stores entries as floats, but the county GeoDataFrame stores FIPS codes as strings, with the first two digits in the `STATEFP` column and the last three digits in the `COUNTYFP` column. Thus, you will need to add these two columns together and then convert them into floats so they can be merged with the `fips` column in the `nytimes` DataFrame.

Drop the regions from the county GeoDataFrame with the same `STATEFP` codes as in Problem 2. Also, make sure to change the CRS of the county GeoDataFrame to EPSG:5071 *before* you merge the two DataFrames (this will make the code run much faster).

Plot the cases from March 21, 2020, and then plot your state outline map from Problem 2 on top of that (with a CRS of EPSG:5071). Include a colorbar using the arguments `legend=True` and `cmap='plasma_r'` in the `plot` function. Finally, print out the name of the county with the most cases on March 21, 2020, along with its case count.

Hint: every state should have multiple covid cases.

[a]Source: `https://raw.githubusercontent.com/nytimes/covid-19-data/master/us-counties.csv`

Logarithmic Plotting Techniques

The color scheme of a graph can also help to communicate information clearly. A good list of available colormaps can be found at `https://matplotlib.org/3.2.1/gallery/color/colormap_reference.html`. Note also that you can reverse any colormap by adding `_r` to the end. The following example demonstrates some plotting features, using country GDP as in Figure 11.3.

```
>>> fig, ax = plt.subplots(1, figsize=(10,4))
>>> world.plot(column='GDP_MD', cmap='plasma_r',
...         ax=ax, legend=True, edgecolor='gray')

# Add title and remove axis tick marks
>>> ax.set_title('GDP on Linear Scale')
>>> ax.set_yticks([])
>>> ax.set_xticks([])
>>> plt.show()
```

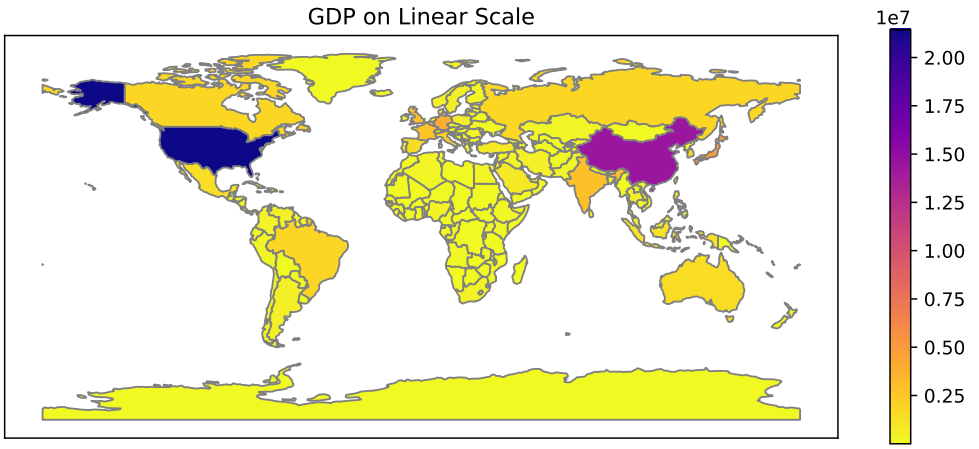


Figure 11.4: World map showing country GDP

Sometimes data can be much more informative when plotted on a logarithmic scale. See how the world map changes when we add a `norm` argument in the code below. Depending on the purpose of the graph, Figure 11.5 may be more informative than Figure 11.4.

```
>>> from matplotlib.colors import LogNorm
>>> from matplotlib.cm import ScalarMappable
>>> fig, ax = plt.subplots(1, figsize=(10,7))

# Set the norm using data bounds
>>> data = world.GDP_MD
>>> norm = LogNorm(vmin=min(data), vmax=max(data))

# Plot the graph using the norm
>>> world.plot(column='GDP_MD', cmap='plasma_r', ax=ax,
...         edgecolor='gray', norm=norm)

# Create a custom colorbar
>>> cbar = fig.colorbar(ScalarMappable(norm=norm, cmap='plasma_r'),
...         ax=ax, orientation='horizontal', pad=0, label='GDP')

>>> ax.set_title('GDP on a Log Scale')
>>> ax.set_yticks([])
>>> ax.set_xticks([])
>>> plt.show()
```

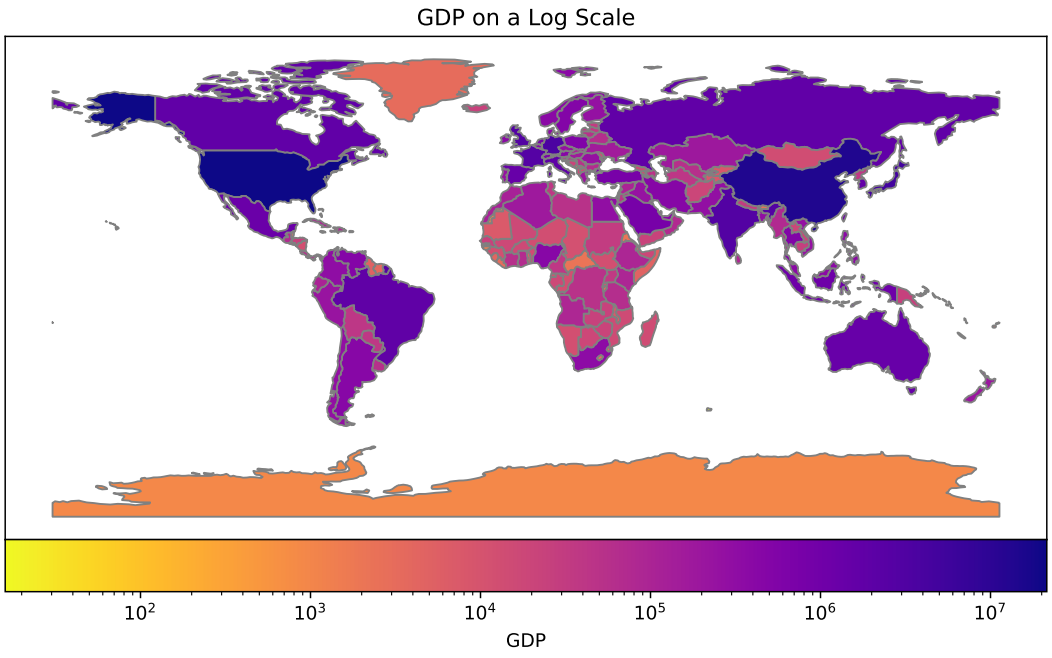


Figure 11.5: World map showing country GDP using a log scale

Problem 4. As in Problem 3, plot your state outline map from Problem 2 on top of a map of Covid-19 cases from March 21, 2020 (each with a CRS of EPSG:5071). This time, however, use a log scale. Pick a good colormap (the counties with the most cases should generally be darkest) and be sure to display a colorbar.

Problem 5. In this problem, you will create an animation of the spread of Covid-19 through US counties from January 21, 2020, through June 21, 2020. You will use the same GeoDataFrame you used in Problems 3 and 4 (with a CRS of EPSG:5071). Use a log scale and a good colormap, and be sure that you're using the same norm and colorbar for the whole animation.

As a reminder, below is a summary of what you will need in order to animate this map. You may also find it helpful to refer to the animation section included with the Volume 4 lab manual.

1. Set up your figure and norm. Be sure to use the highest case count for your `vmax` so that the scale remains uniform.
2. Write your `update` function. This should plot the cases from a given day as well as the state boundaries.
3. Set up your colorbar. Do this outside the `update` function to avoid adding a new colorbar each day.
4. Create a `FuncAnimation` object. Check to make sure everything displays properly before you save it.
5. Save the animation to a file, and embed it into the notebook.

12 Data Cleaning

Lab Objective: *The quality of a data analysis or model is limited by the quality of the data used. In this lab we learn techniques for cleaning data, creating features, and determining feature importance.*

Data cleaning is the process of identifying and correcting bad data. This could be data that is missing, duplicated, irrelevant, inconsistent, incorrect, in the wrong format, or otherwise does not make sense. Though it can be tedious, data cleaning is the most important step of data analysis. Without accurate and legitimate data, any results or conclusions are suspect and may be incorrect.

We will demonstrate common issues with data and how to correct them using the following dataset. It consists of family members and some basic details.

```
# Example dataset
>>> df = pd.read_csv('toy_dataset.csv')

>>> df
          Name  Age        name         DOB Marital_Status
0     John Doe   30        john  01/01/2010       Divorcee
1     Jane Doe   29        jane  12/02/1990       Divorced
2   Jill smith   40         NaN  03/04/1980        married
3   Jill smith   40        jill  03/04/1980        married
4   jack smith  100        jack    4/4/1980       marrieed
5  Jenny Smith    5         NaN  05/05/2015            NaN
6  JAmes Smith    2         NaN  20/06/2018         single
7        Rover    2         NaN  05/05/2018            NaN



   Height  Weight    Marriage_Len      Spouse
0    72.0     175               5         NaN
1     5.5     125               5    John Doe
2    64.0     120              10  Jack Smith
3    64.0     120             NaN  jack smith
4     1.8     220              10  jill smith
5   105.0      40             NaN         NaN
```

```
6    27.0      25  Not Applicable         NaN
7    36.0      50             NaN         NaN
```

Data Type

We can check the data type in Pandas using `dtype`. A `dytpe` of `object` means that the data in that column contains either strings or mixed `dtypes`. These fields should be investigated to determine if they contain mixed datatypes. In our toy example, we would expect that `Marriage_Len` is numerical, so an object `dtype` is suspicious. Looking at the data, we see that `James` has `Not Applicable`, which is a string.

```
# Check validity of data
# Check Data Types
>>> df.dtypes
Name               object
Age                 int64
name               object
DOB                object
Marital_Status     object
Height            float64
Weight              int64
Marriage_Len       object
Spouse             object
dtype: object
```

Duplicates

Duplicates can be easily identified in Pandas using the `duplicated()` function. When no parameters are passed, it returns a DataFrame of the first duplicates. We can identify rows that are duplicated in only some columns by passing in the column names. The `keep` parameter has three possible values, first, last, and False. False keeps all duplicated values, while first and last keep only the first and last instances, respectively.

```
# Display duplicated rows
>>> df[df.duplicated()]
Empty DataFrame
Columns: [Name, Age, name, DOB, Marital_Status, Height, Weight, Marriage_Len, ↩
    Spouse]
Index: []

# Display rows that have duplicates in some columns
>>> df[df.duplicated(['Name','DOB','Marital_Status'],keep=False)]
         Name  Age  name         DOB Marital_Status  Height  Weight ↩
            Marriage_Len      Spouse
2  Jill smith   40   NaN  03/04/1980        married    64.0     120          ↩
    10  Jack Smith
```

```
3  Jill smith   40  jill  03/04/1980        married    64.0     120          ↩
    NaN  jack smith
```

Range

We can check the range of values in a numeric column using the `min` and `max` attributes. If a column corresponds to the temperature in Salt Lake City, measured in degrees Farenheit, then a value over 110 or below 0 should make you suspicious, since those would be extreme values for Salt Lake City. In fact, checking the all-time temperature records for Salt Lake shows that the values in this column should never be more than 107 and never less than -30. Any values outside that range are almost certainly errors and should probably be reset to NaN, unless you have special information that allows you to impute more accurate values.

Other options for looking at the values include line plots, histograms, and boxplots. Some other useful Pandas commands for evaluating the breadth of a dataset include `df.nunique()` (which returns a series giving the name of each column and the number of unique values in each column), `pd.unique()` (which returns an array of the unique values in a series), and `value_counts()` (which counts the number of instances of each unique value in a column, like a histogram).

```
# Count the number of unique values in each column
>>> df.nunique()
Name              7
Age               6
name              4
DOB               7
Marital_Status    5
Height            7
Weight            7
Marriage_Len      3
Spouse            4
dtype: int64


# Print the unique Marital_Status values
>>> pd.unique(df['Marital_Status'])
array(['Divorcee', 'Divorced', 'married', 'marrieed', nan, 'single'],
      dtype=object)


# Count the number of each Marital_Status values
>>> df['Marital_Status'].value_counts()
Marital_Status
married     2
Divorcee    1
Divorced    1
marrieed    1
single      1
Name: count, dtype: int64
```

Missing Data

The percentage of missing data is the completeness of the data. All uncleaned data will have missing values, but datasets with large amounts of missing data, or lots of missing data in key columns, are not going to be as useful. Pandas has several functions to help identify and count missing values. In Pandas, all missing data is considered a `NaN` and does not affect the dtype of a column. `df.isna()` returns a boolean DataFrame indicating whether each value is missing. `df.notnull()` returns a boolean DataFrame with `True` where a value is not missing. Information on how to deal with missing data is described in later sections.

```
# Count number of missing data points in each column
>>> df.isna().sum()
Name              0
Age               0
name              4
DOB               0
Marital_Status    2
Height            0
Weight            0
Marriage_Len      3
Spouse            4
dtype: int64
```

Consistency

Consistency measures how cohesive the data is, both within the dataset and across multiple datasets. For example, in our toy dataset `Jack Smith` is 100 years old, but his birth year is 1980. Data is inconsistent across datasets when the data points should be the same and are different. This could be due to incorrect entries or syntax errors.

It is also important to be consistent in units of measure. Looking at the `Height` column in our dataset, we see values ranging from 1.8 to 105. This is likely the result of different units of measure. It is also important to be consistent across multiple datasets.

All features should also have a consistent type and standard formatting (like capitalization). Syntax errors should be fixed, and white space at the beginning and ends of strings should be removed. Some data might need to be padded so that it's all the same length.

Method	Description
series.str.lower()	Convert to all lower case
series.str.upper()	Convert to all upper case
series.str.strip()	Remove all leading and trailing white space
series.str.lstrip()	Remove leading white space
series.str.replace(" ","")	Remove all spaces
series.str.pad()	Pad strings

Table 12.1: Pandas String Formatting Methods

Problem 1. The `g_t_results.csv` file is a set of parent-reported scores on their child's Gifted and Talented tests. The two tests, `OLSAT` and `NNAT`, are used by NYC to determine if children are qualified for gifted programs. The OLSAT Verbal has 16 questions for Kindergarteners and 30 questions for first, second, and third graders. The NNAT has 48 questions. Each test assigns 1 point to each question asked (so there are no non integer scores). Using this dataset, answer the following questions.

1. What column has the highest number of null values and what percent of its values are null? Print the answer as a tuple with (column name, percentage). Make sure the second value is a percent.
2. List the columns that should be numeric that aren't. Print the answer as a tuple.
3. How many third graders have scores outside the valid range for the OLSAT Verbal Score? Print the answer
4. How many data values are missing (NaN)? Print the number.

Cleaning

There are many aspects and methods of cleaning; here are a few ways of doing so.

Unwanted Data

Removing unwanted data typically falls into two categories, duplicated data and irrelevant data. Irrelevant data consists of observations that don't fit the specific problem you are trying to solve or don't have enough variation to affect the model. We can drop duplicated data using the `duplicated()` function described above with `drop()` or `drop_duplicates()`.

Missing Data

Some commonly suggested methods for handling data are removing the missing data and setting the missing values to some value based on other observations. However, missing data can be informative and removing or replacing missing data erases that information. Removing missing values from a dataset might result in losing significant amounts of data or even in a less accurate model. Retaining the missing values can help increase accuracy.
We have several options to deal with missing data:

- Dropping missing data is the easiest method. Dropping rows or columns should only be done if there is less than $90\% - 95\%$ of the data available. If dropping missing data is inappropriate, you may instead choose to estimate the missing values which can be done by solving the mean, mode, median, randomly choosing from a distribution, linear regression, or hot-decking, to name a few.
- Hot-decking is when you fill in the data based on similar observations. It can be applied to numerical and categorical data, unlike many of the other options listed above. Sequential hot-decking sorts the column with missing data based on an auxiliary column and then fills in the data with the value from the next available data point. K-Nearest Neighbors can also be used to identify similar data points.

- The last option is to flag the data as missing. This retains the information from missing data and removes the missing data (by replacing it). For categorical data, simply replace the data with a new category. For numerical data, we can fill the missing data with 0, or some value that makes sense, and add an indicator variable for missing data.

```
## Replace missing data
import numpy as np

# Add an indicator column based on missing Marriage_Len
>>> df['missing_ML'] =df['Marriage_Len'].isna()

# Fill in all missing data with 0
>>> df['Marriage_Len'] = df['Marriage_Len'].fillna(0)

# Change all other NaNs to missing
>>> df = df.fillna('missing')

# Change Not Applicable row to NaNs
>>> df = df.replace('Not Applicable',np.nan)

# Drop rows with NaNs
>>> df = df.dropna()


>>> df
          Name  Age     name         DOB Marital_Status
0     John Doe   30     john  01/01/2010       Divorcee
1     Jane Doe   29     jane  12/02/1990       Divorced
2   Jill smith   40  missing  03/04/1980        married
3   Jill smith   40     jill  03/04/1980        married
4   jack smith  100     jack    4/4/1980       marrieed
5  Jenny Smith    5  missing  05/05/2015        missing
7        Rover    2  missing  05/05/2018        missing

    Height  Weight  Marriage_Len       Spouse  missing_ML
0     72.0     175             5      missing       False
1     5.5      125             5     John Doe       False
2     64.0     120            10   Jack Smith       False
3     64.0     120             0   jack smith        True
4      1.8     220            10   jill smith       False
5    105.0      40             0      missing        True
7     36.0      50             0      missing        True
```

Nonnumerical Values Misencoded as Numbers

Missing data should always be stored in a form that cannot accidentally be incorporated into the model. This is typically done by storing missing values as NaN. Some algorithms will not run on data with NaN values, in which case you may choose to fill missing data with a string `'missing'`. Many datasets have recorded missing values with a 0 or some other number. You should verify that this does not occur in your dataset.

Categorical data are also often encoded as numerical values. These values should not be left as numbers that can be computed with. For example, postal codes are shorthand for locations, and there is no numerical meaning to the code. It makes no sense to add, subtract, or multiply postal codes, so it is important to not do so. It is good practice to convert this kind of non-numeric data into strings or other data types that cannot be computed with.

Ordinal Data

Ordinal data is data that has a meaningful order but the differences between the values aren't consistent or meaningful. For example, a survey question might ask about your level of education, with 1 being high-school graduate, 2 bachelor's degree, 3 master's degree, and 4 doctoral degree. These values are called ordinal data because it is meaningful to talk about an answer of 1 being less than an answer of 2, but the difference between 1 and 2 is not necessarily the same as the difference between 3 and 4. Subtracting, taking the average, and other algebraic methods do not make a lot of sense with ordinal data.

Problem 2. `imdb.csv` contains a small set of information about 99 movies. Valid movies for this dataset should be longer than 30 minutes long, should have a positive `imdb_score`, and have a `title_year` after 2000.

Clean the data set by doing the following in order:

1. Remove duplicate rows by dropping the first or last. Print the shape of the dataframe after removing the rows.
2. Drop all rows that contain missing data. Print the shape of the dataframe after removing the rows.
3. Remove rows that have data outside of the valid data ranges described above and explain briefly how you determined your ranges for each column.
4. Identify and drop columns with three or fewer different values. Print a tuple with the names of the columns dropped.
5. Convert the titles to all lower case.

Print the first five rows of your dataframe.

Feature Engineering

Constructing new features is called *feature engineering*. Once new features are created, we can analyze how much a model depends on each feature. Features with low importance probably do not contribute much and could potentially be removed.

Discrete Fourier transforms and wavelet decomposition often reveal important properties of data collected over time (*called time-series*), like sound, video, economic indicators, etc. In many such settings it is useful to engineer new features from a wavelet decomposition, the DFT, or some other function of the data.

Engineering for Categorical Variables

Categorical features are those that take only a finite number of values, and usually no categorical value has a numerical meaning, even if it happens to be number. For example in an election dataset, the names of the candidates in the race are categorical, and there is no numerical meaning (neither ordering nor size) to numbers assigned to candidates based soley on their names.

Consider the following election data.

Ballot number	For Governor	For President
001	Herbert	Romney
002	Cooke	Romney
003	Cooke	Obama
004	Herbert	Romney
005	Herbert	Romney
006	Cooke	Stein

A common mistake occurs when someone assigns a number to each categorical entry (say 1 for Cooke, 2 for Herbert, 3 for Romney, etc.). While this assignment is not, in itself, inherently incorrect, it is incorrect to use the value of this number in a statistical model. Whenever you encounter categorical data that is encoded numerically like this, immediately change it to non-numerical form ("Cooke," "Herbert," "Romney,"...) or apply *one-hot encoding* or *dummy variable encoding*. [1] To do this construct a new feature for every possible value of the categorical variable, and assign the value 1 to that feature if the variable takes that value and zero otherwise. Pandas makes one-hot encoding simple:

```
# one-hot encoding
df = pd.get_dummies(df, columns=['For President'])
```

The previous dataset, when the presidential race is one-hot encoded, becomes

Ballot number	Governor	Romney	Obama	Stein
001	Herbert	1	0	0
002	Cooke	1	0	0
003	Cooke	0	1	0
004	Herbert	1	0	0
005	Herbert	1	0	0
006	Cooke	0	0	1

[1] Yes, these are silly names, but they are the most common names for it. Unfortunately, it is probably too late to change these now.

Note that the sum of the terms of the one-hot encoding in each row is 1, corresponding to the fact that every ballot had exactly one presidential candidate.

When the gubernatorial race is also one-hot encoded, this becomes

Ballot number	Cooke	Herbert	Romney	Obama	Stein
001	0	1	1	0	0
002	1	0	1	0	0
003	1	0	0	1	0
004	0	1	1	0	0
005	0	1	1	0	0
006	1	0	0	0	1

Now the sum of the terms of the one-hot encodings in each row is 2, corresponding to the fact that every ballot had two names—one gubernatorial candidate and one presidential candidate.

Summing the columns of the one-hot-encoded data gives the total number of votes for the candidate of that column. So the numerical values in the one-hot encodings are actually numerically meaningful, and summing the entries gives meaningful information. One-hot encoding also avoids the pitfalls of incorrectly using numerical proxies for categorical data.

The main disadvantage of one-hot encoding is that it is an inefficient representation of the data. If there are C categories and n datapoints, a one-hot encoding takes an $n \times 1$-dimensional feature and turns it into an $n \times C$ sparse matrix. But there are ways to store these data efficiently and still maintain the benefits of the one-hot encoding.

Achtung!

When performing linear regression, it is good practice to add a constant column to your dataset and to remove one column of the one-hot encoding of each categorical variable. This is done because the constant column is a linear combination of the one-hot encoded columns causing the matrix to fail to be invertible and can cause identifiability problems.

The standard way to deal with this is to remove one column of the one-hot embedding for each categorical variable. For example, with the elections dataset above, we could remove the Cooke (governor variable) and Romney (presidential variable) columns. Doing that means that in the new dataset a row sum of 0 corresponds to a ballot with a vote for Cooke and a vote for Romney, while a 1 in any column indicates how the ballot differed from the base choice of Cooke and Romney.

When using pandas, you can drop the first column of a one-hot encoding by passing in `drop_first=True`.

Problem 3. Load `housing.csv` into a dataframe with `index_col=0`. Descriptions of the features are in `housing_data_description.txt` for your convenience. The goal is to construct a regression model that predicts SalePrice using the other features of the dataset. Do this as follows:

1. Identify and handle the missing data. Hint: Dropping every row with some missing data is not a good choice because it gives you an empty dataframe. What can you do instead?

2. Create two new features:
   (a) `Remodeled`: Whether or not a house has been remodeled with a `Y` if it has been remodeled, or a `N` if it has not.
   (b) `TotalPorch`: Using the 5 different porch/deck columns, create a new column that provides the total square footage of all the decks and porches for each house.
3. Identify the variable with nonnumerical values that are misencoded as numbers. One-hot encode it. (Hint: don't forget to remove one of the encoded columns to prevent collinearity with the constant column).
4. Add a constant column to the dataframe.
5. Save a copy of the dataframe.
6. Choose four categorical features that seem very important in predicting SalePrice. One-hot encode these features, and remove all other categorical features.
7. Run an OLS (Ordinary Least Squares) regression on your model.

Print a list of the ten features that have the highest coefficients in your model. Print the summary for the dataset as well.

To run an OLS model in python, use the following code.

```
import statsmodels.api as sm

# In our case, y is SalesPrice, and X is the rest of the dataset.
>>> results = sm.OLS(y, X).fit()

# Print the summary
>>> results.summary()

# Convert the summary table to a dataframe
>>> results_as_html = results.summary().tables[1].as_html()
>>> result_df = pd.read_html(results_as_html, header=0, index_col=0)[0]
```

Problem 4. Using the copy of the dataframe you created in Problem 3, one-hot encode all the categorical variables. Print the shape of your database, and Run OLS.
Print the ten features that have the highest coefficient in your model and the summary. Write a couple of sentences discussing which model is better and why.

13 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

Imagine that you are in charge of constructing a very large building. You could, in theory, do all of the work yourself, but that would take so long that it simply would be impractical. Instead, you hire workers, who collectively can work on many parts of the building at once. Managing who does what task takes some effort, but the overall effect is that the building will be constructed many times faster than if only one person was working on it. This is the essential idea behind parallel computing.

A *serial* program is executed one line at a time in a single process. This is analogous to a single person creating a building. Since modern computers have multiple processor cores, serial programs only use a fraction of the computer's available resources. This is beneficial for smooth multitasking on a personal computer because multiple programs can run at once without interrupting each other.

For smaller computations, running serially is fine. However, some tasks are large enough that running serially could take days, months, or in some cases years. In these cases it is beneficial to devote all of a computer's resources (or the resources of many computers) to a single program by running it in *parallel*. Each processor can run part of the program on some of the inputs, and the results can be combined together afterwards. In theory, using N processors at once can allow the computation to run N times faster. Even though communication and coordination overhead prevents the improvement from being quite that good, the difference is still substantial.

A *computer cluster* or *supercomputer* is essentially a group of regular computers that share their processors and memory. There are several common architectures that are used for parallel computing, and each architecture has a different protocol for sharing memory, processors, and tasks 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.

In this lab, we will explore the usage and capabilities of parallel computing using Python's iPyParallel package. iPyParallel can be installed using pip:

```
$ pip install ipyparallel==8.6.1
```

Use version 8.6.1.

The iPyParallel Architecture

There are three main parts of the iPyParallel architecture:

- *Client*: The main 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 processors. Each engine is like a separate Python terminal, each with its own namespace and computing resources.

Essentially, a Python program using iPyParallel creates a `Client` object connected to the cluster that allows it to send tasks to the cluster and retreive their results. The engines run the tasks, and the controller manages which engines run which tasks.

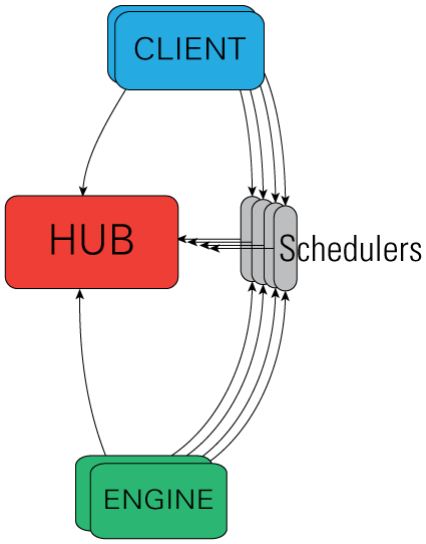


Figure 13.1: An outline of the iPyParallel architecture.

Setting up an iPyParallel Cluster

Before being able to use iPyParallel in a script or interpreter, it is necessarty to start an iPyParallel cluster. We demonstrate here how to use a single machine with multiple processor cores as a cluster. Establishing a cluster on multiple machines requires additional setup, which is detailed in the Additional Material section. The following commands initialize parts or all of a cluster when run 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 stopped 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 connections from clients and engines. Engines will connect automatically to the controller when they start running. 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 as there are cores to maximize efficiency.

> **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")
```

```
# Make sure to include client.close() after each function or else the test ↩
    driver will time out
client.close()
```

Before continuing, set the `DirectView` you are using to use blocking:

```
>>> dview.block = True
```

This affects the way that functions called using the `DirectView` return their values. Using blocking makes the process simpler, so we will use it initially. What blocking is will be explained later.

Problem 1. Write a function that initializes a `Client` object, creates a `DirectView` (with blocking) with all available engines, and imports `scipy.sparse` as `sparse` on all engines. Return the `DirectView`. Note: Make sure to include client.close() after EVERY function or else the test driver will time out.

Managing Engine Namespaces

We now discuss how to handle namespaces within each engine, beginning with setting and retrieving variables.

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 also has a shortcut via indexing.

```
# 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]                # Output from each engine

# 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]
```

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* and is usually used on the results of processing. This method of distributing a dataset and collecting the results is common for processing large data sets using parallelization.

```
>>> 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 either of the execution methods.

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.

```
>>> 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 main program sends tasks to the controller, and then waits for all of the engines to finish their tasks before continuing (the controller "blocks" the program's execution). This mode is usually best for problems in which each node is performing the same task.
- *Non-Blocking*: The main program sends tasks to the controller, and then continues without waiting for responses. Instead of the results, functions return an `AsyncResult` object that can be used to check the execution status and eventually retrieve the actual result.

Whether a function uses blocking is determined by default by the `block` attribute of the `DirectView` The execution methods `execute()`, `apply()`, and `map()`, as well as `push()`, `pull()`, `scatter()`, and `gather()`, each have a keyword argument `block` that can instead be used to specify whether or not to using blocking. 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.
# Both methods produced a list, although the contents are different
>>> block_results[10]  # This list holds actual result values from each engine.
[3.833061790352166,
4.8943956129713335,
4.268791758626886,
4.73533677711277]

>>> responses[10]           # This list holds AsyncResult objects.
<AsyncResult: <lambda>:finished>
# We can get the actual results by using the get() method of each AsyncResult
>>> %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.

>>> nonblock_results[10]    # This list also holds actual result values
[5.652608204341693,
4.984164642641558,
4.686288406810953,
5.275735658763963]
```

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 checkpoints to retrieve answers and enough memory to store response objects.

Class Method	Description
`wait(timeout)`	Wait until the result is available or until `timeout` seconds pass.
`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 13.1: All information from `https://ipyparallel.readthedocs.io/en/latest/details.html#AsyncResult`.

Table 13.1 details the methods of the `AsyncResult` object.

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

Problem 2. Write a function that accepts an integer n. Instruct each engine to make n draws from the standard normal distribution, then hand back the mean, minimum, and maximum draws to the client. Return the results in three lists.

If you have four engines running, your results should resemble the following:

```
>>> means, mins, maxs = problem3(1000000)
>>> means
[0.0031776784, -0.0058112042, 0.0012574772, -0.0059655951]
>>> mins
[-4.1508589, -4.3848019, -4.1313324, -4.2826519]
>>> maxs
[4.0388107, 4.3664958, 4.2060184, 4.3391623]
```

Problem 3. Use your function from Problem 2 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 serially. Make n draws and then calculate and record the statistics, but use 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.

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 feasible to solve through parallel computing because solving them serially would take too long. With some of these 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 4. The *trapezoid rule* is a simple technique for numerical integration:

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

where $a = x_1 < x_2 < \ldots < x_N = b$ and $h = x_{n+1} - x_n$ for each n. See Figure 13.2.

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, divide the points among all available processors and run the trapezoid rule on each portion simultaneously. Be sure that the intervals for each processor share endpoints so that the subintervals are not skipped, resulting in rather large amounts of error. 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 (consider coding the problem in serial to test your function).

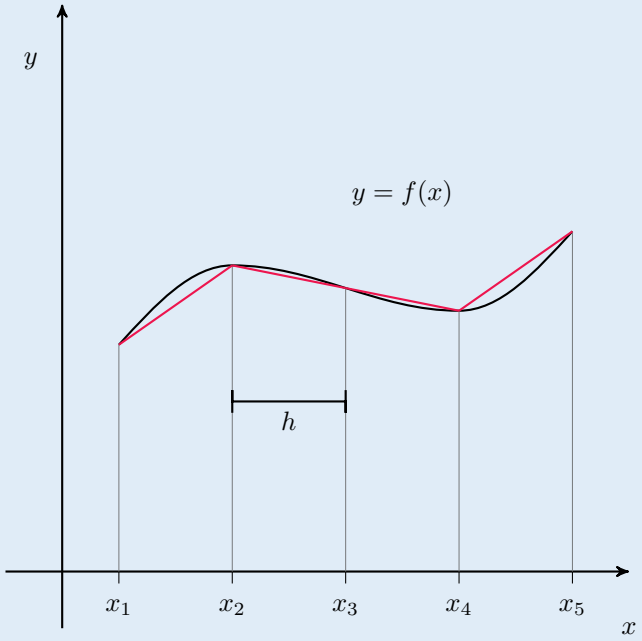


Figure 13.2: A depiction of the trapezoid rule with uniform partitioning.

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 the MPI lab.

Additional Material

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 easily 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]]
```

14 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. 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.

To install mpi4py on a Linux (or WSL) machine, you will need to execute the following commands:

```
$ sudo apt-get install libopenmpi-dev
$ pip install mpi4py
```

And to install mpi4py on a Mac, you will need to execute the following commands:

```
$ brew install openmpi
$ brew install mpi4py
```

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

examplecode/hello.py

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

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

> **Achtung!**
>
> Note that some systems may require `python3`, `py`, or a specific version like `python3.10` instead of `python` as shown above. Check the python environment that your IDE uses to ensure your code runs correctly.

Achtung!

Some systems may not have enough slots to run the desired number of processes. In this case, the `--oversubscribe` flag must be used to ignore the number of availalbe slots when running multiple processes. Therefore, the command will look like

```
$  mpiexec --oversubscribe -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(f"The number of processes is {SIZE}.")
```

examplecode/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(f"My rank is {RANK}.")
```

examplecode/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.

```python
#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))
```

examplecode/separateCode.py

Problem 1. Write a program which determines the rank `n` of the calling process and prints "Hello from process `n`" if `n` is even and "Goodbye from process `n`" if `n` is odd.

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()`.

```python
#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(f"Process 1: Sending: {num_buffer} to process 0.")
    COMM.Send(num_buffer, dest=0)
    print("Process 1: Message sent.")
if RANK == 0:  # This process receives a value from process 1
    num_buffer = np.zeros(1)
    print(f"Process 0: Waiting for the message... current {num_buffer=}.")
    COMM.Recv(num_buffer, source=1)
    print(f"Process 0: Message received! {num_buffer=}.")
```

examplecode/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. For example, a string must be packaged inside an array before it can be passed. The `tag` object can help distinguish between data if multiple pieces of data are being sent/received by the same processes.

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])
```

examplecode/Send_example.py

Problem 2. Write a script that runs on two (and only two!) processes and passes a random numpy array of length `n` from the root process to process 1. Write it so that the user passes in the value of `n` as a command-line argument. The following code demonstrates how to access command-line arguments.

```
from sys import argv

# The first command line argument is saved as n.
n = int(argv[1])
```

In process 1, instantiate a zero array of length `n`, then print this array clearly labeled under process 1. Then have the root process generate a random array, and print this array clearly labeled under the root process. Then send the randomly generated array in the root process to process 1, and print the array clearly labeled under process 1. The output should reflect the following for $n = 4$ (make sure to follow the output formatting exactly).

```
$ mpiexec -n 2 python problem2.py 4

Process 1: Before checking mailbox: vec=[ 0.  0.  0.  0.]
Process 0: Sent: vec=[ 0.03162613  0.38340242  0.27480538  0.56390755]
Process 1: Received: vec=[ 0.03162613  0.38340242  0.27480538  0.56390755]
```

Hint: if the number of processes is not 2, you can abort the program with `COMM.Abort()`.

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.

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`).

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 script should work with any number of processes. Does the order in which `Send()` and `Recv()` are called matter?

Generate an initial random value in each process. Print each initial value clearly labeled under its process (match the formatting given below). Then send the values to the next process as explained above, and print the value each process ends with clearly labeled. The output should reflect the following (note that both the values and order will vary).

```
$ mpiexec -n 2 python problem3.py

Process 1 started with [ 0.79711384]
Process 1 received [ 0.54029085]
Process 0 started with [ 0.54029085]
Process 0 received [ 0.79711384]

$ mpiexec -n 3 python problem3.py

Process 2 started with [ 0.99893055]
Process 0 started with [ 0.6304739]
Process 1 started with [ 0.28834079]
Process 1 received [ 0.6304739]
Process 2 received [ 0.28834079]
Process 0 received [ 0.99893055]
```

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))
```

examplecode/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). The root process should finally print the volume estimate given by the entire set of $(r-1)N$ points, with the dimension of the unit ball clearly labeled.

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 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. The output of a 4 process estimate of U_2 with 2000 draws should reflect the following.

```
$ mpiexec -n 4 python problem4.py 2 2000

Volume of 2-D unit ball: 3.13266666667
```

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.

15 Apache Spark

Lab Objective: *Dealing with massive amounts of data often requires parallelization and cluster computing; Apache Spark is an industry standard for doing just that. In this lab we introduce the basics of PySpark, Spark's Python API, including data structures, syntax, and use cases. Finally, we conclude with a brief introduction to the Spark Machine Learning Package.*

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 package. You must install Spark, along with the supporting tools like Java, on your local machine for PySpark to work. This will include ensuring that both Java and Spark are included in the environment variable `PATH`.[1] Installation of PySpark for local execution or remote connection to an existing cluster can be accomplished by running

```
$ pip install pyspark <= 3.4.1
```

To install Java on WSL or another Linux system, run the following commands in the terminal:

```
$ sudo apt-get install openjdk-8-jdk
```

For Mac M1 users, installing Java will be a little trickier.[2] With Homebrew installed, you'll need to install Maven by running

```
$ brew install maven
```

You will then navigate to this link to download the JDK 8 .dmg file. Enter the following details if the link doesn't automatically populate them for you:

[1] See the Apache Spark configuration instructions for detailed installation instructions

[2] Full instructions can be found at https://dev.to/shane/configure-m1-mac-to-use-jdk8-with-maven-4b4g

```
Java Version - Java 8 (LTS)
Operating System - macOS
Architecture - ARM 64-bit
Java Package - JDK
```

Then install the .dmg file. Now, add the `JAVA_Home` environment variable to the `~/.zshrc` file (with vim or some other file editor) by adding the following line to the file:

```
export JAVA_HOME=/Library/Java/JavaVirtualMachines/zulu-8.jdk/Contents/Home/jre
```

Restart your terminal, and you should be all set. If this process takes longer than a few minutes, you should stop and finish this lab on one of the lab machines.

PySpark

One major benefit of using PySpark is the ability to run it in an interactive environment. One such option is the interactive Spark shell that comes prepackaged with PySpark. 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 use `help(function)`; to exit the shell simply run `quit()`.

In the interactive shell, the `SparkSession` object - the main entrypoint to all Spark functionality - is available by default as `spark`. When running Spark in a standard Python script (or in IPython) you need to define this object explicitly. The code box below outlines how to do this. It is standard practice to name your `SparkSession` object `spark`.

> **Achtung!**
>
> It is important that when you are finished with a `SparkSession` you should end it by calling `spark.stop()`. For this lab, in each problem you will need to instantiate a new `SparkSession` in each problem and then end the session at the end of the function using `spark.stop()`.

> **Note**
>
> While the interactive shell is very robust, it may be easier to learn Spark in an environment that you are more familiar with (like IPython). To do so, just use the code given below. Help can be accessed in the usual way for your environment. Just remember to `stop()` the `SparkSession`!

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

# instantiate your SparkSession object
>>> spark = SparkSession\
...         .builder\
...         .appName("app_name")\
...         .getOrCreate()
```

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

NOTE

The syntax

```
>>> spark = SparkSession\
...          .builder\
...          .appName("app_name")\
...          .getOrCreate()
```

is somewhat unusual. While this code can be written on a single line, it is often more readable to break it up when dealing with many chained operations; this is standard styling for Spark. Note that you *cannot* write a comment after a line continuation character '\'.

Resilient Distributed Datasets

The most fundamental data structure used in Apache Spark is the Resilient Distributed Dataset (RDD). RDDs are immutable distributed collections of objects. They are *resilient* because performing an operation on one RDD produces a *new* RDD without altering the original; if something goes wrong, you can always go back to your original RDD and restart. They are *distributed* because the data resides in logical partitions across multiple machines. While RDDs can be difficult to work with, they offer the most granular control of all the Spark data structures.

There are two main ways of creating RDDs. The first is reading a file directly into Spark and the second is parallelizing an existing collection (list, numpy array, pandas dataframe, etc.). We will use the Titanic dataset[3] in most of the examples throughout this lab. The example below shows various ways to load the Titanic dataset as an RDD.

```
# initialize your SparkSession object
>>> spark = SparkSession\
...          .builder\
...          .appName("app_name")\
...          .getOrCreate()

# load the data directly into an RDD
>>> titanic = spark.sparkContext.textFile('titanic.csv')

# the file is of the format
# Pclass,Survived,Name,Sex,Age,Sibsp,Parch,Ticket,Fare
# Survived | Class | Name | Sex | Age | Siblings/Spouses Aboard | Parents/↩
    Children Aboard | Fare
```

[3] https://web.stanford.edu/class/archive/cs/cs109/cs109.1166/problem12.html

```
>>> titanic.take(2)
['0,3,Mr. Owen Harris Braund,male,22,1,0,7.25',
 '1,1,Mrs. John Bradley (Florence Briggs Thayer) Cumings,female,38,1,0,71.283']

# note that each element is a single string - not particularly useful
# one option is to first load the data into a numpy array
>>> np_titanic = np.loadtxt('titanic.csv', delimiter=',', dtype=list)

# use sparkContext to parallelize the data into 4 partitions
>>> titanic_parallelize = spark.sparkContext.parallelize(np_titanic, 4)

>>> titanic_parallelize.take(2)
[array(['0', '3', ..., 'male', '22', '1', '0', '7.25'], dtype=object),
 array(['1', '1', ..., 'female', '38', '1', '0', '71.2833'], dtype=object)]

 # end SparkSession
 >>> spark.stop()
```

Achtung!

Because Apache Spark partitions and distributes data, calling for the first `n` objects using the same code (such as `take(n)`) may yield different results on different computers (or even each time you run it on one computer). This is not something you should worry about; it is the result of variation in partitioning and will not affect data analysis.

RDD Operations

Transformations

There are two types of operations you can perform on RDDs: *transformations* and *actions*. Transformations are functions that produce new RDDs from existing ones. Transformations are also lazy; they are not executed until an *action* is performed. This allows Spark to boost performance by optimizing *how* a sequence of transformations is executed at runtime.

One of the most commonly used transformations is the `map(func)`, which creates a new RDD by applying `func` to each element of the current RDD. This function, `func`, can be any callable python function, though it is often implemented as a `lambda` function. Similarly, `flatMap(func)` creates an RDD with the flattened results of `map(func)`.

```
# initialize your SparkSession object
>>> spark = SparkSession\
...          .builder\
...          .appName("app_name")\
...          .getOrCreate()

# use map() to format the data
>>> titanic = spark.sparkContext.textFile('titanic.csv')
>>> titanic.take(2)
```

```
['0,3,Mr. Owen Harris Braund,male,22,1,0,7.25',
 '1,1,Mrs. John Bradley (Florence Briggs Thayer) Cumings,female,38,1,0,71.283']

# apply split(',') to each element of the RDD with map()
>>> titanic.map(lambda row: row.split(','))\
...         .take(2)
[['0', '3', 'Mr. Owen Harris Braund', 'male', '22', '1', '0', '7.25'],
 ['1', '1', ..., 'female', '38', '1', '0', '71.283']]

# compare to flatMap(), which flattens the results of each row
>>> titanic.flatMap(lambda row: row.split(','))\
...         .take(2)
['0', '3']
```

The `filter(func)` transformation returns a new RDD containing only the elements that satisfy `func`. In this case, `func` should be a callable python function that returns a Boolean. The elements of the RDD that evaluate to `True` are included in the new RDD while those that evaluate to `False` are excluded.

```
# create a new RDD containing only the female passengers
>>> titanic = titanic.map(lambda row: row.split(','))
>>> titanic_f = titanic.filter(lambda row: row[3] == 'female')
>>> titanic_f.take(3)
[['1', '1', ..., 'female', '38', '1', '0', '71.2833'],
 ['1', '3', ..., 'female', '26', '0', '0', '7.925'],
 ['1', '1', ..., 'female', '35', '1', '0', '53.1']]
```

Note

A great transformation to help validate or explore your dataset is `distinct()`. This will return a new RDD containing only the distinct elements of the original. In the case of the Titanic dataset, if you did not know how many classes there were, you could do the following:

```
>>> titanic.map(lambda row: row[1])\
...         .distinct()\
...         .collect()
['1', '3', '2']
```

Spark Command	Transformation
`map(f)`	Returns a new RDD by applying `f` to each element of this RDD
`flatMap(f)`	Same as `map(f)`, except the results are flattened
`filter(f)`	Returns a new RDD containing only the elements that satisfy `f`
`distinct()`	Returns a new RDD containing the distinct elements of the original
`reduceByKey(f)`	Takes an RDD of `(key, val)` pairs and merges the values for each `key` using an associative and commutative reduce function `f`
`sortBy(f)`	Sorts this RDD by the given function `f`
`sortByKey(f)`	Sorts an RDD assumed to consist of `(key, val)` pairs by the given function `f`
`groupBy(f)`	Returns a new RDD of groups of items based on `f`
`groupByKey()`	Takes an RDD of `(key, val)` pairs and returns a new RDD with `(key, (val1, val2, ...))` pairs

```
# the following counts the number of passengers in each class
# note that this isn't necessarily the best way to do this

# create a new RDD of (pclass, 1) elements to count occurances
>>> pclass = titanic.map(lambda row: (row[1], 1))
>>> pclass.take(5)
[('3', 1), ('1', 1), ('3', 1), ('1', 1), ('3', 1)]

# count the members of each class
>>> pclass = pclass.reduceByKey(lambda x, y: x + y)
>>> pclass.collect()
[('3', 487), ('1', 216), ('2', 184)]

# sort by number of passengers in each class, ascending order
>>> pclass.sortBy(lambda row: row[1]).collect()
[('2', 184), ('1', 216), ('3', 487)]

# end SparkSession
>>> spark.stop()
```

Achtung!

Note that you must use `.collect()` to extract data from an RDD. Using `.collect()` will return an array.

Problem 1. Write a function that accepts the name of a text file with default `filename='huck_finn.txt'`.[a] Load the file as a PySpark RDD, and count the number of occurrences of each word. Sort the words by count, in descending order, and return a list of the `(word, count)` pairs for the 20 most used words. The data does not need to be cleaned.

Hint: to ensure that your function doesn't consider an empty string `""` to be a word, make sure to split each line with `.split()` instead of on a space with `.split(' ')`.

[a] https://www.gutenberg.org/files/76/76-0.txt

Actions

Actions are operations that return non-RDD objects. Two of the most common actions, `take(n)` and `collect()`, have already been seen above. The key difference between the two is that `take(n)` returns the first `n` elements from one (or more) partition(s) while `collect()` returns the contents of the entire RDD. When working with small datasets this may not be an issue, but for larger datasets running `collect()` can be very expensive.

Another important action is `reduce(func)`. Generally, `reduce()` combines (reduces) the data in each row of the RDD using `func` to produce some useful output. Note that `func` *must* be an associative and commutative binary operation; otherwise the results will vary depending on partitioning.

```
# create an RDD with the first million integers in 4 partitions
>>> ints = spark.sparkContext.parallelize(range(1, 1000001), 4)
# [1, 2, 3, 4, 5, ..., 1000000]
# sum the first one million integers
>>> ints.reduce(lambda x, y: x + y)
500000500000

# create a new RDD containing only survival data
>>> survived = titanic.map(lambda row: int(row[0]))
>>> survived.take(5)
[0, 1, 1, 1, 0]

# find total number of survivors
>>> survived.reduce(lambda x, y: x + y)
500
```

Spark Command	Action
`take(n)`	returns the first `n` elements of an RDD
`collect()`	returns the entire contents of an RDD
`reduce(f)`	merges the values of an RDD using an associative and commutative operator `f`
`count()`	returns the number of elements in the RDD
`min(); max(); mean()`	returns the minimum, maximum, or mean of the RDD, respectively
`sum()`	adds the elements in the RDD and returns the result
`saveAsTextFile(path)`	saves the RDD as a collection of text files (one for each partition) in the directory specified
`foreach(f)`	immediately applies `f` to each element of the RDD; not to be confused with `map()`, `foreach()` is useful for saving data somewhere not natively supported by PySpark

Problem 2. 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. Your function should accept two keyword arguments: `n=10**5` and `parts=6`. Use `n*parts` sample points and partition your RDD with `parts` partitions. Return your estimate.

[a] See Example 7.1.1 in the Volume 2 textbook

DataFrames

While RDDs offer granular control, they can be slower than their Scala and Java counterparts when implemented in Python. The solution to this was the creation of a new data structure: Spark DataFrames. Just like RDDs, DataFrames are immutable distributed collections of objects; however, unlike RDDs, DataFrames are organized into named (and typed) columns. In this way they are conceptually similar to a relational database (or a pandas DataFrame).

The most important difference between a relational database and Spark DataFrames is in the execution of transformations and actions. When working with DataFrames, Spark's Catalyst Optimizer creates and optimizes a logical execution plan before sending any instructions to the drivers. After the logical plan has been formed, an optimal physical plan is created and executed. This provides significant performance boosts, especially when working with massive amounts of data. Since the Catalyst Optimizer functions the same across all language APIs, DataFrames bring performance parity to all of Spark's APIs.

Spark SQL and DataFrames

Creating a DataFrame from an existing text, csv, or JSON file is generally easier than creating an RDD. The DataFrame API also has arguments to deal with file headers or to automatically infer the schema.

```
# note that you should initialize your spark object first
# load the titanic dataset using default settings
>>> titanic = spark.read.csv('titanic.csv')
>>> titanic.show(2)
+---+---+--------------------+------+---+---+---+-------+
|_c0|_c1|                 _c2|   _c3|_c4|_c5|_c6|    _c7|
+---+---+--------------------+------+---+---+---+-------+
|  0|  3|Mr. Owen Harris B...|  male| 22|  1|  0|   7.25|
|  1|  1|Mrs. John Bradley...|female| 38|  1|  0|71.2833|
+---+---+--------------------+------+---+---+---+-------+
only showing top 2 rows

# spark.read.csv('titanic.csv', inferSchema=True) will try to infer
# data types for each column

# load the titanic dataset specifying the schema
>>> schema = ('survived INT, pclass INT, name STRING, sex STRING, '
...           'age FLOAT, sibsp INT, parch INT, fare FLOAT'
...           )
>>> titanic = spark.read.csv('titanic.csv', schema=schema)
>>> titanic.show(2)
+--------+------+--------------------+------+---+-----+-----+-------+
|survived|pclass|                name|   sex|age|sibsp|parch|   fare|
+--------+------+--------------------+------+---+-----+-----+-------+
|       0|     3|Mr. Owen Harris B...|  male| 22|    1|    0|   7.25|
|       1|     1|Mrs. John Bradley...|female| 38|    1|    0|71.2833|
+--------+------+--------------------+------+---+-----+-----+-------+
only showing top 2 rows

# for files with headers, the following is convenient
spark.read.csv('my_file.csv', header=True, inferSchema=True)
```

Note

To convert a DataFrame to an RDD use `my_df.rdd`; to convert an RDD to a DataFrame use `spark.createDataFrame(my_rdd)`. You can also use `spark.createDataFrame()` on numpy arrays and pandas DataFrames.

DataFrames can be easily updated, queried, and analyzed using SQL operations. Spark allows you to run queries directly on DataFrames similar to how you perform transformations on RDDs. Additionally, the `pyspark.sql.functions` module contains many additional functions to further analysis. Below are many examples of basic DataFrame operations; further examples involving the `pyspark.sql.functions` module can be found in the additional materials section. Full documentation can be found at https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/index.html.

```
# select data from the survived column
>>> titanic.select(titanic.survived).show(3)  # or titanic.select("survived")
+--------+
|survived|
+--------+
|       0|
|       1|
|       1|
+--------+
only showing top 3 rows

# find all distinct ages of passengers (great for data exploration)
>>> titanic.select("age")\
...         .distinct()\
...         .show(3)
+----+
| age|
+----+
|18.0|
|64.0|
|0.42|
+----+
only showing top 3 rows

# filter the DataFrame for passengers between 20-30 years old (inclusive)
>>> titanic.filter(titanic.age.between(20, 30)).show(3)
+--------+------+--------------------+------+----+-----+-----+------+
|survived|pclass|                name|   sex| age|sibsp|parch|  fare|
+--------+------+--------------------+------+----+-----+-----+------+
|       0|     3|Mr. Owen Harris B...|  male|22.0|    1|    0|  7.25|
|       1|     3|Miss. Laina Heikk...|female|26.0|    0|    0| 7.925|
|       0|     3|     Mr. James Moran|  male|27.0|    0|    0|8.4583|
+--------+------+--------------------+------+----+-----+-----+------+
only showing top 3 rows

# find total fare by pclass (or use .avg('fare') for an average)
>>> titanic.groupBy('pclass')\
...         .sum('fare')\
...         .show()
+------+---------+
|pclass|sum(fare)|
```

```
+------+---------+
|     1| 18177.41|
|     3|  6675.65|
|     2|  3801.84|
+------+---------+

# group and count by age and survival; order age/survival descending
>>> titanic.groupBy("age", "survived").count()\
...         .sort("age", "survived", ascending=False)\
...         .show(2)
+---+--------+-----+
|age|survived|count|
+---+--------+-----+
| 80|       1|    1|
| 74|       0|    1|
+---+--------+-----+
only showing top 2 rows

# join two DataFrames on a specified column (or list of columns)
>>> titanic_cabins.show(3)
+--------------------+-------+
|                name|  cabin|
+--------------------+-------+
|Miss. Elisabeth W...|     B5|
|Master. Hudsen Tr...|C22 C26|
|Miss. Helen Lorai...|C22 C26|
+--------------------+-------+
only showing top 3 rows

>>> titanic.join(titanic_cabins, on='name').show(3)
+--------------------+--------+------+------+----+-----+-----+------+-------+
|                name|survived|pclass|   sex| age|sibsp|parch|  fare|  cabin|
+--------------------+--------+------+------+----+-----+-----+------+-------+
|Miss. Elisabeth W...|       0|     3|  male|22.0|    1|    0|  7.25|     B5|
|Master. Hudsen Tr...|       1|     3|female|26.0|    0|    0| 7.925|C22 C26|
|Miss. Helen Lorai...|       0|     3|  male|27.0|    0|    0|8.4583|C22 C26|
+--------------------+--------+------+------+----+-----+-----+------+-------+
only showing top 3 rows
```

Note

If you prefer to use traditional SQL syntax you can use `spark.sql("SQL QUERY")`. Note that this requires you to first create a temporary view of the DataFrame.

```
# create the temporary view so we can access the table through SQL
>>> titanic.createOrReplaceTempView("titanic")
```

```
# query using SQL syntax
>>> spark.sql("SELECT age, COUNT(*) AS count\
...             FROM titanic\
...             GROUP BY age\
...             ORDER BY age DESC").show(3)
+---+-----+
|age|count|
+---+-----+
| 80|    1|
| 74|    1|
| 71|    2|
+---+-----+
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`
`drop(*cols)`	`DROP`
`join(other, on=None, how=None)`	`JOIN` (join type specified by `how`)
`count()`	`COUNT()`
`sum(*cols)`	`SUM()`
`avg(*cols)` or `mean(*cols)`	`AVG()`
`collect()`	`fetchall()`

Problem 3. Write a function with keyword argument `filename='titanic.csv'`. Load the file into a PySpark DataFrame and find (1) the number of women on-board, (2) the number of men on-board, (3) the survival rate of women, and (4) the survival rate of men. Return these four values in the order given as a tuple of floats.

Problem 4. In this problem, you will be using the `london_income_by_borough.csv` and the `london_crime_by_lsoa.csv` files to visualize the relationship between income and the frequency of crime.[a] The former contains estimated mean and median income data for each London borough, averaged over 2008-2016; the first line of the file is a header with columns `borough`, `mean-08-16`, and `median-08-16`. The latter contains over 13 million lines of crime data, organized by borough and LSOA (Lower Super Output Area) code, for London between 2008 and 2016; the first line of the file is a header, containing the following seven columns:

`lsoa_code`: LSOA code (think area code) where the crime was committed
`borough`: London borough were the crime was committed
`major_category`: major or general category of the crime
`minor_category`: minor or specific category of the crime
`value`: number of occurrences of this crime in the given `lsoa_code`, `month`, and `year`
`year`: year the crime was committed
`month`: month the crime was committed

Write a function that accepts three keyword arguments: `crimefile='london_crime_by_lsoa.csv'`, `incomefile='london_income_by_borough.csv'`, and `major_cat='Robbery'`. Load the two files as PySpark DataFrames. Use them to create a new DataFrame. The new DataFrame will contain a row for each borough and have columns for borough, total number of crimes for the given major category (`major_cat`), and median income. Order the DataFrame by the total number of crimes for `major_cat`, descending. The final DataFrame should have three columns: `borough`, `major_cat_total_crime`, and `median-08-16` (column names may be different).

Convert the DataFrame to a numpy array using `np.array(df.collect())`, and create a scatter plot of the number of `major_cat=` crimes by the median income for each borough. Return the numpy array.

[a] data.london.gov.uk

Machine Learning with Apache Spark

Apache Spark includes a vast and expanding ecosystem to perform machine learning. PySpark's primary machine learning API, `pyspark.ml`, is DataFrame-based.

Here we give a start to finish example using Spark ML to tackle the classic Titanic classification problem.

```
# prepare data
# convert the 'sex' column to binary categorical variable
>>> from pyspark.ml.feature import StringIndexer, OneHotEncoder
>>> sex_binary = StringIndexer(inputCol='sex', outputCol='sex_binary')

# one-hot-encode pclass (Spark automatically drops a column)
>>> onehot = OneHotEncoder(inputCols=['pclass'],
...                                 outputCols=['pclass_onehot'])

# create single features column
```

```
from pyspark.ml.feature import VectorAssembler
features = ['sex_binary', 'pclass_onehot', 'age', 'sibsp', 'parch', 'fare']
features_col = VectorAssembler(inputCols=features, outputCol='features')

# now we create a transformation pipeline to apply the operations above
# this is very similar to the pipeline ecosystem in sklearn
>>> from pyspark.ml import Pipeline
>>> pipeline = Pipeline(stages=[sex_binary, onehot, features_col])
>>> titanic = pipeline.fit(titanic).transform(titanic)

# drop unnecessary columns for cleaner display (note the new columns)
>>> titanic = titanic.drop('pclass', 'name', 'sex')
>>> titanic.show(2)
+--------+----+-----+-----+----+----------+-------------+----------+
|survived| age|sibsp|parch|fare|sex_binary|pclass_onehot|  features|
+--------+----+-----+-----+----+----------+-------------+----------+
|       0|22.0|    1|    0|7.25|       0.0|    (3,[],[])|(8,[4,5...|
|       1|38.0|    1|    0|71.3|       1.0|   (3,[1],...|[0.0,1....|
+--------+----+-----+-----+----+----------+-------------+----------+

# split into train/test sets (75/25)
>>> train, test = titanic.randomSplit([0.75, 0.25], seed=11)

# initialize logistic regression
>>> from pyspark.ml.classification import LogisticRegression
>>> lr = LogisticRegression(labelCol='survived', featuresCol='features')

# run a train-validation-split to fit best elastic net param
# ParamGridBuilder constructs a grid of parameters to search over.
>>> from pyspark.ml.tuning import ParamGridBuilder, TrainValidationSplit
>>> from pyspark.ml.evaluation import MulticlassClassificationEvaluator as MCE
>>> paramGrid = ParamGridBuilder()\
...                 .addGrid(lr.elasticNetParam, [0, 0.5, 1]).build()
# TrainValidationSplit will try all combinations and determine best model using
# the evaluator (see also CrossValidator)
>>> tvs = TrainValidationSplit(estimator=lr,
...                            estimatorParamMaps=paramGrid,
...                            evaluator=MCE(labelCol='survived'),
...                            trainRatio=0.75,
...                            seed=11)

# we train the classifier by fitting our tvs object to the training data
>>> clf = tvs.fit(train)

# use the best fit model to evaluate the test data
>>> results = clf.bestModel.evaluate(test)
>>> results.predictions.select(['survived', 'prediction']).show(5)
+--------+----------+
|survived|prediction|
```

```
+--------+----------+
|       0|       1.0|
|       0|       1.0|
|       0|       1.0|
|       0|       1.0|
|       0|       0.0|
+--------+----------+

# performance information is stored in various attributes of "results"
>>> results.accuracy
0.7527272727272727

>>> results.weightedRecall
0.7527272727272727

>>> results.weightedPrecision
0.751035147726004

# many classifiers do not have this object-oriented interface (yet)
# it isn't much more effort to generate the same statistics for a ↩
    DecisionTreeClassifier, for example
>>> dt_clf = dt_tvs.fit(train) # same process, except for a different paramGrid

# generate predictions - this returns a new DataFrame
>>> preds = clf.bestModel.transform(test)
>>> preds.select('survived', 'probability', 'prediction').show(5)
+--------+-----------+----------+
|survived|probability|prediction|
+--------+-----------+----------+
|       0|  [1.0,0.0]|       0.0|
|       0|  [1.0,0.0]|       0.0|
|       0|  [1.0,0.0]|       0.0|
|       0|  [0.0,1.0]|       1.0|
+--------+-----------+----------+

# initialize evaluator object
>>> dt_eval = MCE(labelCol='survived')
>>> dt_eval.evaluate(preds, {dt_eval.metricName: 'accuracy'})
0.8433179723502304
```

Below is a broad overview of the `pyspark.ml` ecosystem. It should help give you a starting point when looking for a specific functionality.

PySpark ML Module	Module Purpose
`pyspark.ml.feature`	provides functions to transform data into feature vectors
`pyspark.ml.tuning`	grid search, cross validation, and train/validation split functions
`pyspark.ml.evaluation`	tools to compute prediction metrics (accuracy, f1, etc.)
`pyspark.ml.classification`	classification models (logistic regression, SVM, etc.)
`pyspark.ml.clustering`	clustering models (k-means, Gaussian mixture, etc.)
`pyspark.ml.regression`	regression models (linear regression, decision tree regressor, etc.)

Problem 5. Write a function with keyword argument `filename='titanic.csv'`. Load the file into a PySpark DataFrame, and use the `pyspark.ml` package to train a classifier that outperforms the logistic regression each of the three metrics from the example above (`accuracy` , `weightedRecall`, `weightedPrecision`).

Some of Spark's available classifiers are listed below. For complete documentation, visit https://spark.apache.org/docs/l

```
# from pyspark.ml.classification import LinearSVC
#                              DecisionTreeClassifier
#                              GBTClassifier
#                              MultilayerPerceptronClassifier
#                              NaiveBayes
#                              RandomForestClassifier
```

Use `randomSplit([0.75, 0.25], seed=11)` to split your data into train and test sets before fitting the model. Return the `accuracy`, `weightedRecall`, and `weightedPrecision` for your model, in the given order as a tuple.

Hint: to calculate the accuracy of a classifer in PySpark, use `results.accuracy` where `results = (classifier name).bestModel.evaluate(test)`.

Additional Material

Further DataFrame Operations

There are a few other functions built directly on top of DataFrames to further analysis. Additionally, the `pyspark.sql.functions` module expands the available functions significantly.[4]

```
# some immediately accessible functions
# covariance between pclass and fare
>>> titanic.cov('pclass', 'fare')
-22.86289824115662

# summary of statistics for selected columns
>>> titanic.select("pclass", "age", "fare")\
...         .summary().show()
+-------+------------------+------------------+------------------+
|summary|            pclass|               age|              fare|
+-------+------------------+------------------+------------------+
|  count|               887|               887|               887|
|   mean| 2.305524239007892|29.471443066501564|32.305420253026846|
| stddev|0.8366620036697728|14.121908405492908| 49.78204096767521|
|    min|                 1|              0.42|               0.0|
|    25%|                 2|              20.0|             7.925|
|    50%|                 3|              28.0|           14.4542|
|    75%|                 3|              38.0|            31.275|
|    max|                 3|              80.0|          512.3292|
+-------+------------------+------------------+------------------+

# additional functions from the functions module
>>> from pyspark.sql import functions as sqlf

# finding the mean of a column without grouping requires sqlf.avg()
# alias(new_name) allows us to rename the column on the fly
>>> titanic.select(sqlf.avg("age").alias("Average Age")).show()
+------------------+
|       Average Age|
+------------------+
|29.471443066516347|
+------------------+

# use .agg([dict]) on GroupedData to specify [multiple] aggregate
# functions, including those from pyspark.sql.functions
>>> titanic.groupBy('pclass')\
...         .agg({'fare': 'var_samp', 'age': 'stddev'})\
...         .show(3)
+------+------------------+------------------+
|pclass|    var_samp(fare)|       stddev(age)|
+------+------------------+------------------+
```

[4] https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/index.html

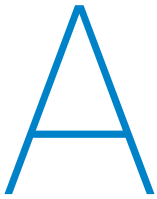
```
|     1| 6143.483042924841|14.183632587264817|
|     3|139.64879027298073|12.095083834183779|
|     2|180.02658999396826|13.756191206499766|
+------+------------------+------------------+

# perform multiple aggregate actions on the same column
>>> titanic.groupBy('pclass')\
...         .agg(sqlf.sum('fare'), sqlf.stddev('fare'))\
...         .show()
+------+------------------+------------------+
|pclass|         sum(fare)| stddev_samp(fare)|
+------+------------------+------------------+
|     1|18177.412506103516| 78.38037409278448|
|     3| 6675.653553009033| 11.81730892686574|
|     2|3801.8417053222656|13.417398778972332|
+------+------------------+------------------+
```

pyspark.sql.functions	Operation
`ceil(col)`	computes the ceiling of each element in `col`
`floor(col)`	computes the floor of each element in `col`
`min(col)`, `max(col)`	returns the minimum/maximum value of `col`
`mean(col)`	returns the average of the values of `col`
`stddev(col)`	returns the unbiased sample standard deviation of `col`
`var_samp(col)`	returns the unbiased variance of the values in `col`
`rand(seed=None)`	generates a random column with i.i.d. samples from $[0, 1]$
`randn(seed=None)`	generates a random column with i.i.d. samples from the standard normal distribution
`exp(col)`	computes the exponential of `col`
`log(arg1, arg2=None)`	returns `arg1`-based logarithm of `arg2`; if there is only one argument, then it returns the natural logarithm
`cos(col)`, `sin(col)`, etc.	computes the given trigonometric or inverse trigonometric (`asin(col)`, etc.) function of `col`

Part II

Appendices

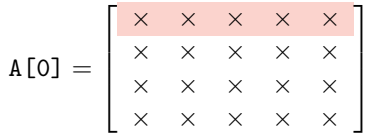


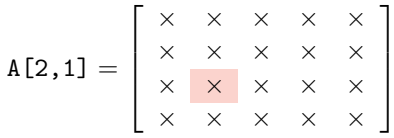
NumPy Visual Guide

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

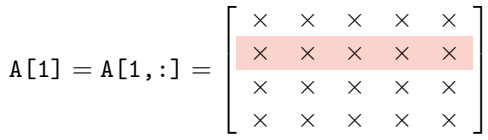
The entries of a 2-D array are the rows of the matrix (as 1-D arrays). To access a single entry, enter the row index, a comma, and the column index. Remember that indexing begins with 0.

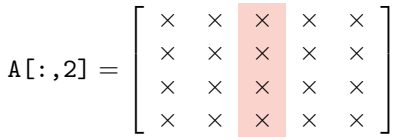


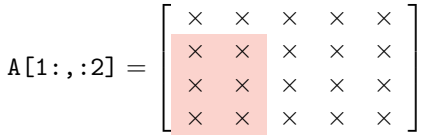


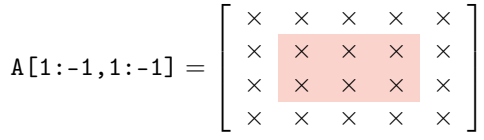
Slicing

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."



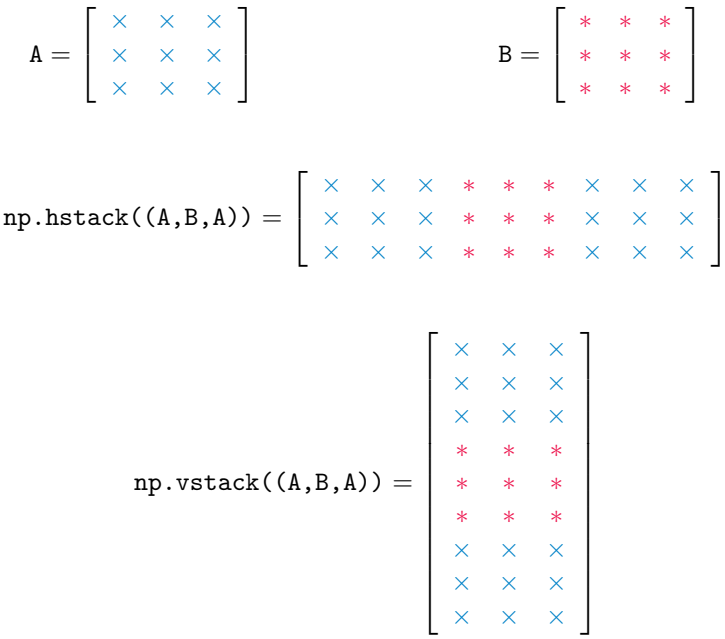




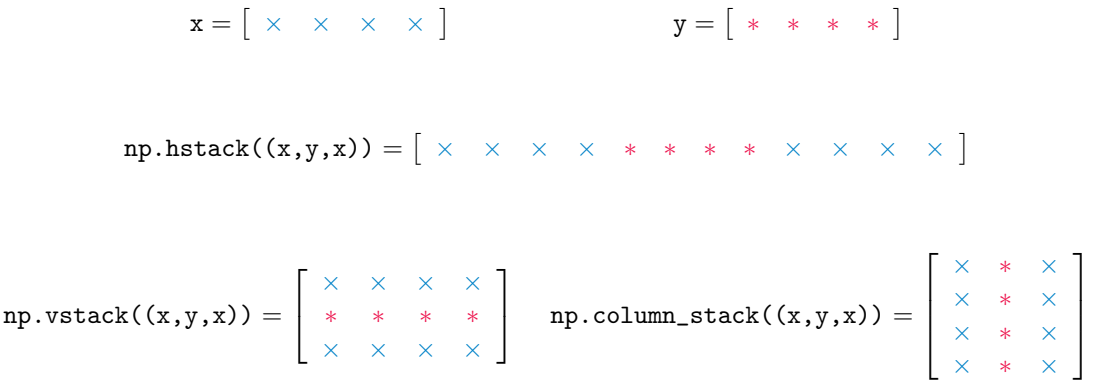


Stacking

`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`.

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

$$\texttt{A.sum(axis=0)} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} = \begin{bmatrix} 4 & 8 & 12 & 16 \end{bmatrix}$$

$$\texttt{A.sum(axis=1)} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} = \begin{bmatrix} 10 & 10 & 10 & 10 \end{bmatrix}$$

B Matplotlib Syntax and Customization Guide

Lab Objective: *The documentation for Matplotlib can be a little difficult to maneuver and basic information is sometimes difficult to find. This appendix condenses and demonstrates some of the more applicable and useful information on plot customizations. It is not intended to be read all at once, but rather to be used as a reference when needed. For an interative introduction to Matplotlib, see the Introduction to Matplotlib lab in Python Essentials. For more details on any specific function, refer to the Matplotlib documentation at* `https://matplotlib.org/`.

Matplotlib Interface

Matplotlib plots are made in a `Figure` object that contains one or more `Axes`, which themselves contain the graphical plotting data. Matplotlib provides two ways to create plots:

1. Call plotting functions directly from the module, such as `plt.plot()`. This will create the plot on whichever `Axes` is currently active.

2. Call plotting functions from an `Axes` object, such as `ax.plot()`. This is particularly useful for complicated plots and for animations.

Table B.1 contains a summary of functions that are used for managing `Figure` and `Axes` objects.

Function	Description
`add_subplot()`	Add a single subplot to the current figure
`axes()`	Add an axes to the current figure
`clf()`	Clear the current figure
`figure()`	Create a new figure or grab an existing figure
`gca()`	Get the current axes
`gcf()`	Get the current figure
`subplot()`	Add a single subplot to the current figure
`subplots()`	Create a figure and add several subplots to it

Table B.1: Basic functions for managing plots.

`Axes` objects are usually managed through the functions `plt.subplot()` and `plt.subplots()`. The function `subplot()` is used as `plt.subplot(nrows, ncols, plot_number)`. Note that if the inputs for `plt.subplot()` are all integers, the commas between the entries can be omitted. For example, `plt.subplot(3,2,2)` can be shortened to `plt.subplot(322)`.

The function `subplots()` is used as `plt.subplots(nrows, ncols)`, and returns a `Figure` object and an array of `Axes`. This array has the shape `(nrows, ncols)`, and can be accessed as any other array. Figure B.1 demonstrates the layout and indexing of subplots.

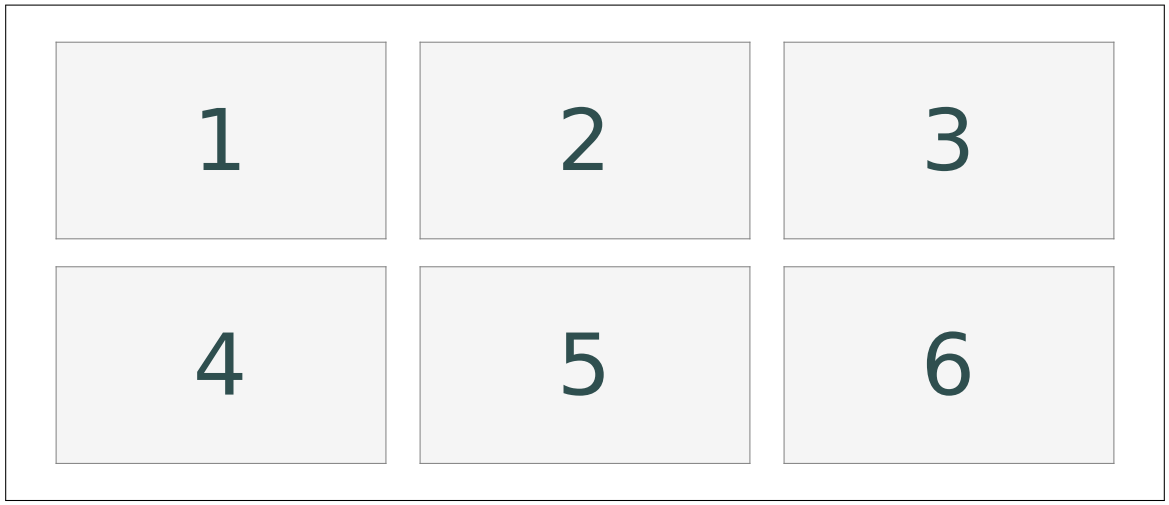


Figure B.1: The layout of subplots with `plt.subplot(2,3,i)` (2 rows, 3 columns), where `i` is the index pictured above. The outer border is the figure that the axes belong to.

The following example demonstrates three equivalent ways of producing a figure with two subplots, arranged next to each other in one row:

```
>>> x = np.linspace(-5, 5, 100)

# 1. Use plt.subplot() to switch the current axes.
>>> plt.subplot(121)
>>> plt.plot(x, 2*x)
>>> plt.subplot(122)
>>> plt.plot(x, x**2)

# 2. Use plt.subplot() to explicitly grab the two subplot axes.
>>> ax1 = plt.subplot(121)
>>> ax1.plot(x, 2*x)
>>> ax2 = plt.subplot(122)
>>> ax2.plot(x, x**2)

# 3. Use plt.subplots() to get the figure and all subplots simultaneously.
>>> fig, axes = plt.subplots(1, 2)
>>> axes[0].plot(x, 2*x)
>>> axes[1].plot(x, x**2)
```

ACHTUNG!

Be careful not to mix up the following similarly-named functions:

1. `plt.axes()` creates a new place to draw on the figure, while `plt.axis()` or `ax.axis()` sets properties of the x- and y-axis in the current axes, such as the x and y limits.
2. `plt.subplot()` (singular) returns a single subplot belonging to the current figure, while `plt.subplots()` (plural) creates a new figure and adds a collection of subplots to it.

Plot Customization

Styles

Matplotlib has a number of built-in styles that can be used to set the default appearance of plots. These can be used via the function `plt.style.use()`; for instance, `plt.style.use("seaborn")` will have Matplotlib use the "seaborn" style for all plots created afterwards. A list of built-in styles can be found at
`https://matplotlib.org/stable/gallery/style_sheets/style_sheets_reference.html`.
The style can also be changed only temporarily using `plt.style.context()` along with a `with` block:

```
with plt.style.context('dark_background'):
    # Any plots created here use the new style
    plt.subplot(1,2,1)
    plt.plot(x, y)
    # ...
# Plots created here are unaffected
plt.subplot(1,2,2)
plt.plot(x, y)
```

Plot layout

Axis properties

Table B.2 gives an overview of some of the functions that may be used to configure the axes of a plot.
The functions `xlim()`, `ylim()`, and `axis()` are used to set one or both of the x and y ranges of the plot. `xlim()` and `ylim()` each accept two arguments, the lower and upper bounds, or a single list of those two numbers. `axis()` accepts a single list consisting, in order, of
`xmin, xmax, ymin, ymax`. Passing `None` instead of one of the numbers to any of these functions will make it not change the corresponding value from what it was. Each of these functions can also be called without any arguments, in which case it will return the current bounds. Note that `axis()` can also be called directly on an `Axes` object, while `xlim()` and `ylim()` cannot.
`axis()` also can be called with a string as its argument, which has several options. The most common is `axis('equal')`, which makes the scale of the x- and y-scales equal (i.e. makes circles circular).

Function	Description
`axis()`	set the x- and y-limits of the plot
`grid()`	add gridlines
`xlim()`	set the limits of the x-axis
`ylim()`	set the limits of the y-axis
`xticks()`	set the location of the tick marks on the x-axis
`yticks()`	set the location of the tick marks on the y-axis
`xscale()`	set the scale type to use on the x-axis
`yscale()`	set the scale type to use on the y-axis
`ax.spines[side].set_position()`	set the location of the given spine
`ax.spines[side].set_color()`	set the color of the given spine
`ax.spines[side].set_visible()`	set whether a spine is visible

Table B.2: Some functions for changing axis properties. `ax` is an `Axes` object.

To use a logarithmic scale on an axis, the functions `xscale("log")` and `yscale("log")` can be used.

The functions `xticks()` and `yticks()` accept a list of tick positions, which the ticks on the corresponding axis are set to. Generally, this works the best when used with `np.linspace()`. This function also optionally accepts a second argument of a list of labels for the ticks. If called with no arguments, the function returns a list of the current tick positions and labels instead.

The spines of a Matplotlib plot are the black border lines around the plot, with the left and bottom ones also being used as the axis lines. To access the spines of a plot, call `ax.spines[side]`, where `ax` is an `Axes` object and `side` is `'top'`, `'bottom'`, `'left'`, or `'right'`. Then, functions can be called on the `Spine` object to configure it.

The function `spine.set_position()` has several ways to specify the position. The two simplest are with the arguments `'center'` and `'zero'`, which place the spine in the center of the subplot or at an x- or y-coordinate of zero, respectively. The others are a passed as a tuple `(position_type, amount)`:

- `'data'`: place the spine at an x- or y-coordinate equal to `amount`.
- `'axes'`: place the spine at the specified `Axes` coordinate, where 0 corresponds to the bottom or left of the subplot, and 1 corresponds to the top or right edge of the subplot.
- `'outward'`: places the spine `amount` pixels outward from the edge of the plot area. A negative value can be used to move it inwards instead.

`spine.set_color()` accepts any of the color formats Matplotlib supports. Alternately, using `set_color('none')` will make the spine not be visible. `spine.set_visible()` can also be used for this purpose.

The following example adjusts the ticks and spine positions to improve the readability of a plot of $\sin(x)$. The result is shown in Figure B.2.

```
>>> x = np.linspace(0,2*np.pi,150)
>>> plt.plot(x, np.sin(x))
>>> plt.title(r"$y=\sin(x)$")

#Set the ticks to multiples of pi/2, make nice labels
>>> ticks = np.pi / 2 * np.array([0,1,2,3,4])
```

```
>>> tick_labels = ["$0$", r"$\frac{\pi}{2}$", r"$\pi$", r"$\frac{3\pi}{2}$",
...                 r"$2\pi$"]
>>> plt.xticks(ticks, tick_labels)

#Move the bottom spine to zero, remove the top and right ones
>>> ax = plt.gca()
>>> ax.spines['bottom'].set_position('zero')
>>> ax.spines['right'].set_color('none')
>>> ax.spines['top'].set_color('none')

>>> plt.show()
```

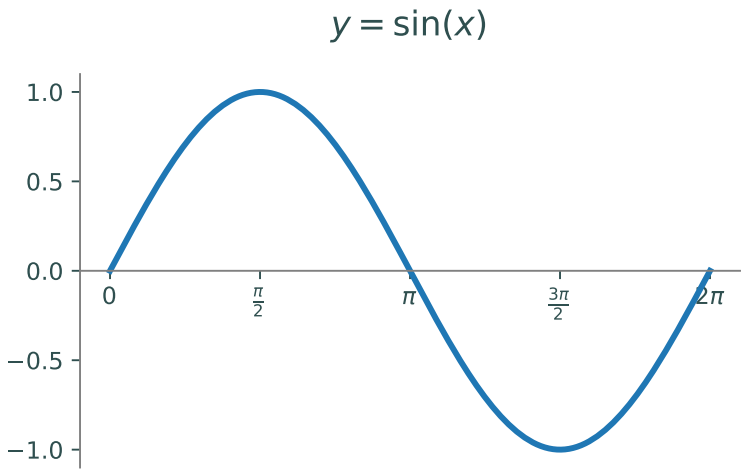


Figure B.2: Plot of $y = \sin(x)$ with axes modified for clarity

Plot Layout

The position and spacing of all subplots within a figure can be modified using the function `plt.subplots_adjust()`. This function accepts up to six keyword arguments that change different aspects of the spacing. `left`, `right`, `top`, and `bottom` are used to adjust the rectangle around all of the subplots. In the coordinates used, 0 corresponds to the bottom or left edge of the figure, and 1 corresponds to the top or right edge of the figure. `hspace` and `wspace` set the vertical and horizontal spacing, respectively, between subplots. The units for these are in fractions of the average height and width of all subplots in the figure. If more fine control is desired, the position of individual `Axes` objects can also be changed using `ax.get_position()` and `ax.set_position()`. The size of the figure can be configured using the `figsize` argument when creating a figure:

```
>>> plt.figure(figsize=(12,8))
```

Note that many environments will scale the figure to fill the available space. Even so, changing the figure size can still be used to change the aspect ratio as well as the relative size of plot elements. The following example uses `subplots_adjust()` to create space for a legend outside of the plotting space. The result is shown in Figure B.3.

```
#Generate data
>>> x1 = np.random.normal(-1, 1.0, size=60)
>>> y1 = np.random.normal(-1, 1.5, size=60)
>>> x2 = np.random.normal(2.0, 1.0, size=60)
>>> y2 = np.random.normal(-1.5, 1.5, size=60)
>>> x3 = np.random.normal(0.5, 1.5, size=60)
>>> y3 = np.random.normal(2.5, 1.5, size=60)

#Make the figure wider
>>> fig = plt.figure(figsize=(5,3))

#Plot the data
>>> plt.plot(x1, y1, 'r.', label="Dataset 1")
>>> plt.plot(x2, y2, 'g.', label="Dataset 2")
>>> plt.plot(x3, y3, 'b.', label="Dataset 3")

#Create a legend to the left of the plot
>>> lspace = 0.35
>>> plt.subplots_adjust(left=lspace)
#Put the legend at the left edge of the figure
>>> plt.legend(loc=(-lspace/(1-lspace),0.6))
>>> plt.show()
```

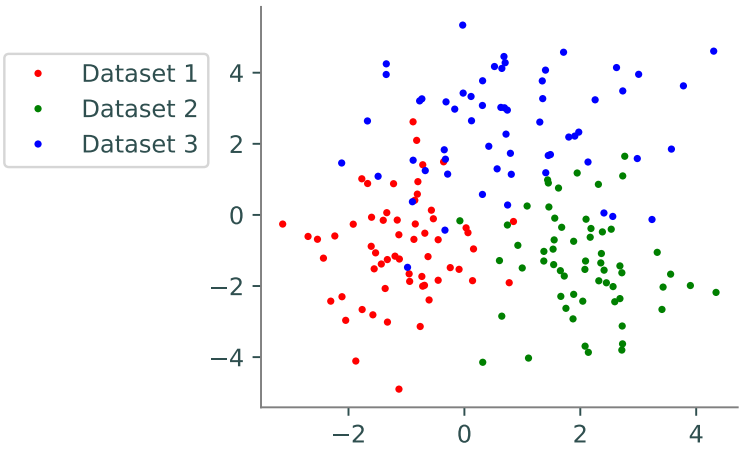


Figure B.3: Example of repositioning axes.

Colors

The color that a plotting function uses is specified by either the `c` or `color` keyword arguments; for most functions, these can be used interchangeably. There are many ways to specific colors. The most simple is to use one of the basic colors, listed in Table B.3. Colors can also be specified using an RGB tuple such as `(0.0, 0.4, 1.0)`, a hex string such as `"0000FF"`, or a CSS color name like `"DarkOliveGreen"` or `"FireBrick"`. A full list of named colors that Matplotlib supports can be found at `https://matplotlib.org/stable/gallery/color/named_colors.html`. If no color is specified for a plot, Matplotlib automatically assigns it one from the default color cycle.

Code	Color
`'b'`	blue
`'g'`	green
`'r'`	red
`'c'`	cyan
`'m'`	magenta

Code	Color
`'y'`	yellow
`'k'`	black
`'w'`	white
`'C0'` - `'C9'`	Default colors

Table B.3: Basic colors available in Matplotlib

Plotting functions also accept an `alpha` keyword argument, which can be used to set the transparency. A value of `1.0` corresponds to fully opaque, and `0.0` corresponds to fully transparent.

The following example demonstrates different ways of specifying colors:

```
#Using a basic color
>>> plt.plot(x, y, 'r')
#Using a hexadecimal string
>>> plt.plot(x, y, color='FF0080')
#Using an RGB tuple
>>> plt.plot(x, y, color=(1, 0.5, 0))
#Using a named color
>>> plt.plot(x, y, color='navy')
```

Colormaps

Certain plotting functions, such as heatmaps and contour plots, accept a colormap rather than a single color. A full list of colormaps available in Matplotlib can be found at `https://matplotlib.org/stable/gallery/color/colormap_reference.html`. Some of the more commonly used ones are `"viridis"`, `"magma"`, and `"coolwarm"`. A colorbar can be added by calling `plt.colorbar()` after creating the plot.

Sometimes, using a logarithmic scale for the coloring is more informative. To do this, pass a `matplotlib.colors.LogNorm` object as the `norm` keyword argument:

```
# Create a heatmap with logarithmic color scaling
>>> from matplotlib.colors import LogNorm
>>> plt.pcolormesh(X, Y, Z, cmap='viridis', norm=LogNorm())
```

Function	Description	Usage
`annotate()`	adds a commentary at a given point on the plot	annotate('text',(x,y))
`arrow()`	draws an arrow from a given point on the plot	arrow(x,y,dx,dy)
`colorbar()`	Create a colorbar	colorbar()
`legend()`	Place a legend in the plot	legend(loc='best')
`text()`	Add text at a given position on the plot	text(x,y,'text')
`title()`	Add a title to the plot	title('text')
`suptitle()`	Add a title to the figure	suptitle('text')
`xlabel()`	Add a label to the x-axis	xlabel('text')
`ylabel()`	Add a label to the y-axis	ylabel('text')

Table B.4: Text and annotation functions in Matplotlib

Text and Annotations

Matplotlib has several ways to add text and other annotations to a plot, some of which are listed in Table B.4. The color and size of the text in most of these functions can be adjusted with the `color` and `fontsize` keyword arguments.

Matplotlib also supports formatting text with LaTeX, a system for creating technical documents.[1] To do so, use an `r` before the string quotation mark and surround the text with dollar signs. This is particularly useful when the text contains a mathematical expression. For example, the following line of code will make the title of the plot be $\frac{1}{2}\sin(x^2)$:

```
>>> plt.title(r"$\frac{1}{2}\sin(x^2)$")
```

The function `legend()` can be used to add a legend to a plot. Its optional `loc` keyword argument specifies where to place the legend within the subplot. It defaults to `'best'`, which will cause Matplotlib to place it in whichever location overlaps with the fewest drawn objects. The other locations this function accepts are `'upper right'`, `'upper left'`, `'lower left'`, `'lower right'`, `'center left'`, `'center right'`, `'lower center'`, `'upper center'`, and `'center'`. Alternately, a tuple of `(x,y)` can be passed as this argument, and the bottom-left corner of the legend will be placed at that location. The point (0,0) corresponds to the bottom-left of the current subplot, and (1,1) corresponds to the top-right. This can be used to place the legend outside of the subplot, although care should be taken that it does not go outside the figure, which may require manually repositioning the subplots.

The labels the legend uses for each curve or scatterplot are specified with the `label` keyword argument when plotting the object. Note that `legend()` can also be called with non-keyword arguments to set the labels, although it is less confusing to set them when plotting.

The following example demonstrates creating a legend:

```
>>> x = np.linspace(0,2*np.pi,250)

# Plot sin(x), cos(x), and -sin(x)
# The label argument will be used as its label in the legend.
>>> plt.plot(x, np.sin(x), 'r', label=r'$\sin(x)$')
>>> plt.plot(x, np.cos(x), 'g', label=r'$\cos(x)$')
>>> plt.plot(x, -np.sin(x), 'b', label=r'$-\sin(x)$')
```

[1] See `http://www.latex-project.org/` for more information.

```
# Create the legend
>>> plt.legend()
```

Line and marker styles

Matplotlib supports a large number of line and marker styles for line and scatter plots, which are listed in Table B.5.

character	description	character	description
-	solid line style	3	tri_left marker
--	dashed line style	4	tri_right marker
-.	dash-dot line style	s	square marker
:	dotted line style	p	pentagon marker
.	point marker	*	star marker
,	pixel marker	h	hexagon1 marker
o	circle marker	H	hexagon2 marker
v	triangle_down marker	+	plus marker
^	triangle_up marker	x	x marker
<	triangle_left marker	D	diamond marker
>	triangle_right marker	d	thin_diamond marker
1	tri_down marker	|	vline marker
2	tri_up marker	_	hline marker

Table B.5: Available line and marker styles in Maplotlib.

The function `plot()` has several ways to specify this argument; the simplest is to pass it as the third positional argument. The `marker` and `linestyle` keyword arguments can also be used. The size of these can be modified using `markersize` and `linewidth`. Note that by specifying a marker style but no line style, `plot()` can be used to make a scatter plot. It is also possible to use both a marker style and a line style. To set the marker using `scatter()`, use the `marker` keyword argument, with `s` being used to change the size.

The following code demonstrates specifying marker and line styles. The results are shown in Figure B.4.

```
#Use dashed lines:
>>> plt.plot(x, y, '--')
#Use only dots:
>>> plt.plot(x, y, '.')
#Use dots with a normal line:
>>> plt.plot(x, y, '.-')
#scatter() uses the marker keyword:
>>> plt.scatter(x, y, marker='+')

#With plot(), the color to use can also be specified in the same string.
#Order usually doesn't matter.
#Use red dots:
>>> plt.plot(x, y, '.r')
```

```
#Equivalent:
>>> plt.plot(x, y, 'r.')

#To change the size:
>>> plt.plot(x, y, 'v-', linewidth=1, markersize=15)
>>> plt.scatter(x, y, marker='+', s=12)
```

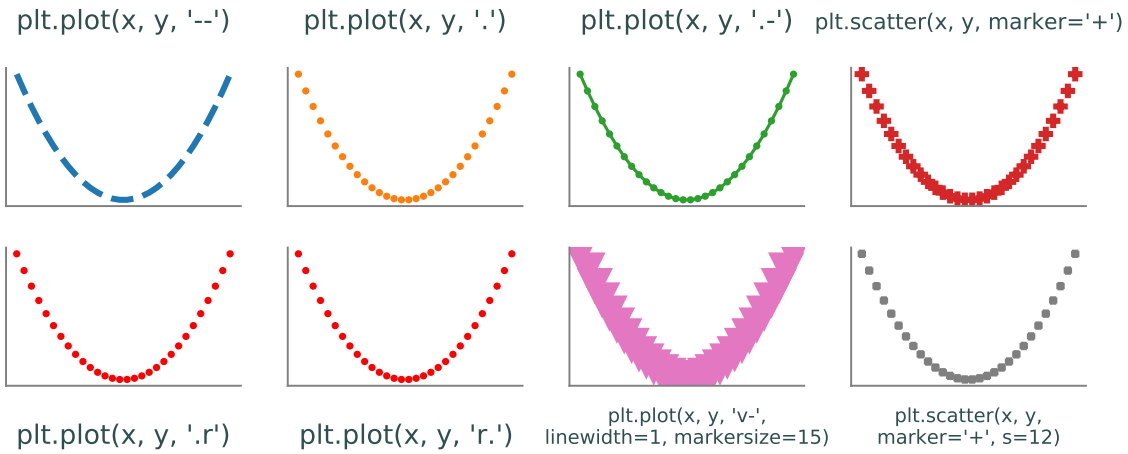


Figure B.4: Examples of setting line and marker styles.

Plot Types

Matplotlib has functions for creaing many different types of plots, many of which are listed in Table B.6. This section gives details on using certain groups of these functions.

Function	Description	Usage
`bar`	makes a bar graph	bar(x,height)
`barh`	makes a horizontal bar graph	barh(y,width)
`boxplots`	makes one or more boxplots	boxplots(data)
`contour`	makes a contour plot	contour(X,Y,Z)
`contourf`	makes a filled contour plot	contourf(X,Y,Z)
`imshow`	shows an image	imshow(image)
`fill`	plots lines with shading under the curve	fill(x,y)
`fill_between`	plots lines with shading between two given y values	fill_between(x,y1, y2=0)
`hexbin`	creates a hexbin plot	hexbin(x,y)
`hist`	plots a histogram from data	hist(data)
`pcolormesh`	makes a heatmap	pcolormesh(X,Y,Z)
`pie`	makes a pie chart	pie(x)
`plot`	plots lines and data on standard axes	plot(x,y)
`plot_surface`	plot a surface in 3-D space	plot_surface(X,Y,Z)
`polar`	plots lines and data on polar axes	polar(theta,r)
`loglog`	plots lines and data on logarithmic x and y axes	loglog(x,y)
`scatter`	plots data in a scatterplot	scatter(x,y)
`semilogx`	plots lines and data with a log scaled x axis	semilogx(x,y)
`semilogy`	plots lines and data with a log scaled y axis	semilogy(x,y)
`specgram`	makes a spectogram from data	specgram(x)
`spy`	plots the sparsity pattern of a 2D array	spy(Z)
`triplot`	plots triangulation between given points	triplot(x,y)

Table B.6: Some basic plotting functions in Matplotlib.

Line plots

Line plots, the most basic type of plot, are created with the `plot()` function. It accepts two lists of x- and y-values to plot, and optionally a third argument of a string of any combination of the color, line style, and marker style. Note that this method only works with the single-character color codes; to use other colors, use the `color` argument. By specifying only a marker style, this function can also be used to create scatterplots.

There are a number of functions that do essentially the same thing as `plot()` but also change the axis scaling, including `loglog()`, `semilogx()`, `semilogy()`, and `polar`. Each of these functions is used in the same manner as `plot()`, and has identical syntax.

Bar Plots

Bar plots are a way to graph categorical data in an effective way. They are made using the `bar()` function. The most important arguments are the first two that provide the data, `x` and `height`. The first argument is a list of values for each bar, either categorical or numerical; the second argument is a list of numerical values corresponding to the height of each bar. There are other parameters that may be included as well. The `width` argument adjusts the bar widths; this can be done by choosing a single value for all of the bars, or an array to give each bar a unique width. Further, the argument `bottom` allows one to specify where each bar begins on the y-axis. Lastly, the `align` argument can be set to 'center' or 'edge' to align as desired on the x-axis. As with all plots, you can use the `color` keyword to specify any color of your choice. If you desire to make a horizontal bar graph, the syntax follows similarly using the function `barh()`, but with argument names `y`, `width`, `height` and `align`.

Box Plots

A box plot is a way to visualize some simple statistics of a dataset. It plots the minimum, maximum, and median along with the first and third quartiles of the data. This is done by using `boxplot()` with an array of data as the argument. Matplotlib allows you to enter either a one dimensional array for a single box plot, or a 2-dimensional array where it will plot a box plot for each column of the data in the array. Box plots default to having a vertical orientation but can be easily laid out horizontally by setting `vert=False`.

Scatter and hexbin plots

Scatterplots can be created using either `plot()` or `scatter()`. Generally, it is simpler to use `plot()`, although there are some cases where `scatter()` is better. In particular, `scatter()` allows changing the color and size of individual points within a single call to the function. This is done by passing a list of colors or sizes to the `c` or `s` arguments, respectively.

Hexbin plots are an alternative to scatterplots that show the concentration of data in regions rather than the individual points. They can be created with the function `hexbin()`. Like `plot()` and `scatter()`, this function accepts two lists of x- and y-coordinates.

Heatmaps and contour plots

Heatmaps and contour plots are used to visualize 3-D surfaces and complex-valued functions on a flat space. Heatmaps are created using the `pcolormesh()` function. Contour plots are created using `contour()` or `contourf()`, with the latter creating a filled contour plot.

Each of these functions accepts the x-, y-, and z-coordinates as a mesh grid, or 2-D array. To create these, use the function `np.meshgrid()`:

```
>>> x = np.linspace(0,1,100)
>>> y = np.linspace(0,1,80)
>>> X, Y = np.meshgrid(x, y)
```

The z-coordinate can then be computed using the x and y mesh grids.

Note that each of these functions can accept a colormap, using the `cmap` parameter. These plots are sometimes more informative with a logarithmic color scale, which can be used by passing a `matplotlib.colors.LogNorm` object in the `norm` parameter of these functions.

With `pcolormesh()`, it is also necessary to pass `shading='auto'` or `shading='nearest'` to avoid a deprecation error.

The following example demonstrates creating heatmaps and contour plots, using a graph of $z = (x^2 + y)\sin(y)$. The results is shown in Figure B.5

```
>>> from matplotlib.colors import LogNorm

>>> x = np.linspace(-3,3,100)
>>> y = np.linspace(-3,3,100)
>>> X, Y = np.meshgrid(x, y)
>>> Z = (X**2+Y)*np.sin(Y)

#Heatmap
>>> plt.subplot(1,3,1)
>>> plt.pcolormesh(X, Y, Z, cmap='viridis', shading='nearest')
>>> plt.title("Heatmap")

#Contour
>>> plt.subplot(1,3,2)
>>> plt.contour(X, Y, Z, cmap='magma')
>>> plt.title("Contour plot")

#Filled contour
>>> plt.subplot(1,3,3)
>>> plt.contourf(X, Y, Z, cmap='coolwarm')
>>> plt.title("Filled contour plot")
>>> plt.colorbar()

>>> plt.show()
```

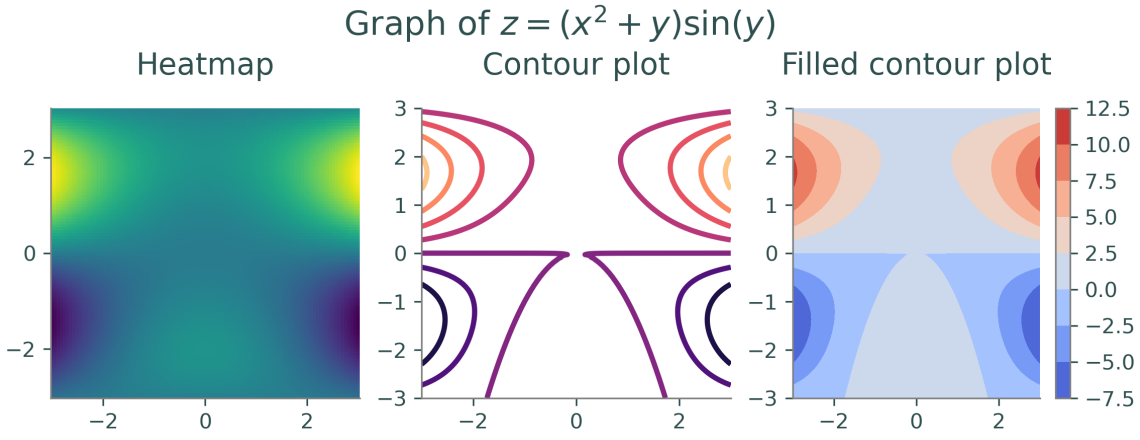


Figure B.5: Example of heatmaps and contour plots.

Showing images

The function `imshow()` is used for showing an image in a plot, and can be used on either grayscale or color images. This function accepts a 2-D $n \times m$ array for a grayscale image, or a 3-D $n \times m \times 3$ array for a color image. If using a grayscale image, you also need to specify `cmap='gray'`, or it will be colored incorrectly.

It is best to also use `axis('equal')` alongside `imshow()`, or the image will most likely be stretched. This function also works best if the images values are in the range $[0, 1]$. Some ways to load images will format their values as integers from 0 to 255, in which case the values in the image array should be scaled before using `imshow()`.

3-D Plotting

Matplotlib can be used to plot curves and surfaces in 3-D space. In order to use 3-D plotting, you need to run the following line:

```
>>> from mpl_toolkits.plot3d import Axes3D
```

The argument `projection='3d'` also must be specified when creating the subplot for the 3-D object:

```
>>> plt.subplot(1,1,1, projection='3d')
```

Curves can be plotted in 3-D space using `plot()`, by passing in three lists of x-, y-, and z-coordinates. Surfaces can be plotted using `ax.plot_surface()`. This function can be used similar to creating contour plots and heatmaps, by obtaining meshes of x- and y- coordinates from `np.meshgrid()` and using those to produce the z-axis. More generally, any three 2-D arrays of meshes corresponding to x-, y-, and z-coordinates can be used. Note that it is necessary to call this function from an Axes object.

The following example demonstrates creating 3-D plots. The results are shown in Figure B.6.

```
#Create a plot of a parametric curve
ax = plt.subplot(1,3,1, projection='3d')
t = np.linspace(0, 4*np.pi, 160)
x = np.cos(t)
y = np.sin(t)
z = t / np.pi
plt.plot(x, y, z, color='b')
plt.title("Helix curve")

#Create a surface plot from np.meshgrid
ax = plt.subplot(1,3,2, projection='3d')
x = np.linspace(-1,1,80)
y = np.linspace(-1,1,80)
X, Y = np.meshgrid(x, y)
Z = X**2 - Y**2
ax.plot_surface(X, Y, Z, color='g')
plt.title(r"Hyperboloid")
```

```
#Create a surface plot less directly
ax = plt.subplot(1,3,3, projection='3d')
theta = np.linspace(-np.pi,np.pi,80)
rho = np.linspace(-np.pi/2,np.pi/2,40)
Theta, Rho = np.meshgrid(theta, rho)
X = np.cos(Theta) * np.cos(Rho)
Y = np.sin(Theta) * np.cos(Rho)
Z = np.sin(Rho)
ax.plot_surface(X, Y, Z, color='r')
plt.title(r"Sphere")

plt.show()
```

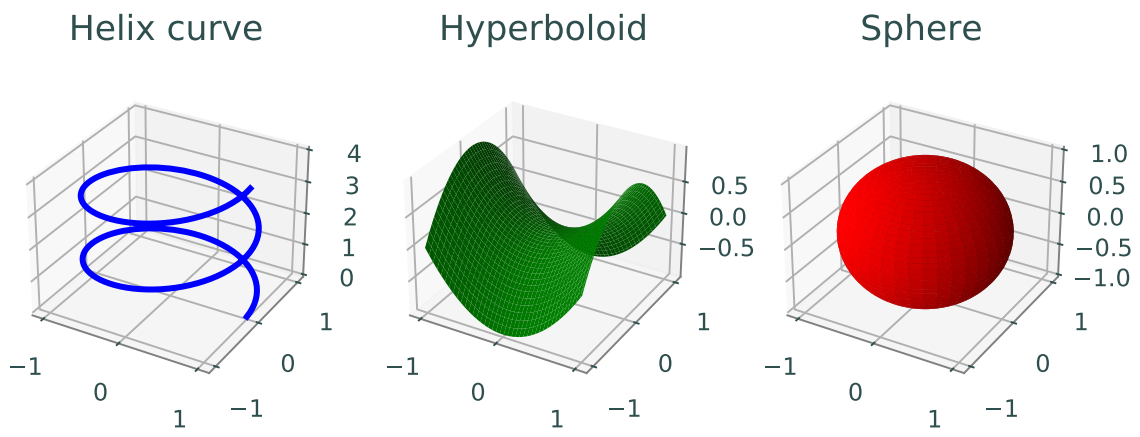


Figure B.6: Examples of 3-D plotting.

Additional Resources

rcParams

The default plotting parameters of Matplotlib can be set individually and with more fine control than styles by using `rcParams`. `rcParams` is a dictionary that can be accessed as either `plt.rcParams` or `matplotlib.rcParams`.
For instance, the resolution of plots can be changed via the `"figure.dpi"` parameter:

```
>>> plt.rcParams["figure.dpi"] = 600
```

A list of parameters that can set via `rcParams` can be found at `https://matplotlib.org/stable/api/matplotlib_configuration_api.html#matplotlib.RcParams`.

Animations

Matplotlib has capabilities for creating animated plots. The Animations lab in Volume 4 has detailed instructions on how to do so.

Matplotlib gallery and tutorials

The Matplotlib documentation has a number of tutorials, found at `https://matplotlib.org/stable/tutorials/index.html`. It also has a large gallery of examples, found at `https://matplotlib.org/stable/gallery/index.html`. Both of these are excellent sources of additional information about ways to use and customize Matplotlib.

Bibliography

[ADH+01] David Ascher, Paul F Dubois, Konrad Hinsen, Jim Hugunin, Travis Oliphant, et al. Numerical python, 2001.

[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.

[VD10] Guido VanRossum and Fred L Drake. *The python language reference*. Python software foundation Amsterdam, Netherlands, 2010.