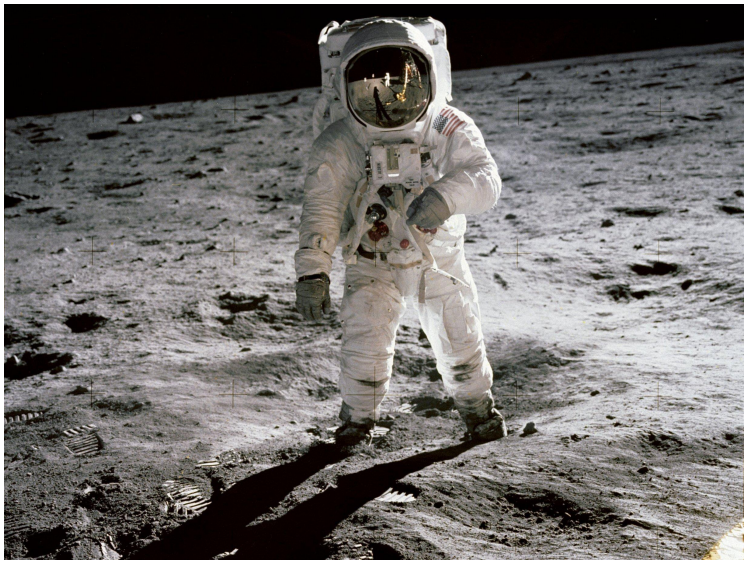
Labs for Foundations of Applied Mathematics

Volume III
Modeling with Uncertainty and Data



List of Contributors

E. Evans
Brigham Young University

R. Evans
Brigham Young University

J. Grout
Drake University

J. Humpherys
Brigham Young University

T. Jarvis
Brigham Young University

J. Whitehead
Brigham Young University

J. Adams
Brigham Young University

J. Bejarano
Brigham Young University

Z. Boyd
Brigham Young University

M. Brown
Brigham Young University

T. Christensen
Brigham Young University

M. Cook
Brigham Young University

R. Dorff
Brigham Young University

B. Ehlert
Brigham Young University

M. Fabiano
Brigham Young University

A. Frandsen
Brigham Young University

K. Finlinson
Brigham Young University

J. Fisher
Brigham Young University

R. Fuhriman
Brigham Young University

S. Giddens
Brigham Young University

C. Gigena
Brigham Young University

M. Graham
Brigham Young University

F. Glines
Brigham Young University

M. Goodwin
Brigham Young University

R. Grout
Brigham Young University

J. Hendricks
Brigham Young University

A. Henriksen
Brigham Young University

I. Henriksen
Brigham Young University

C. Hettinger
Brigham Young University

S. Horst
Brigham Young University

K. Jacobson
Brigham Young University

J. Leete
Brigham Young University

J. Lytle
Brigham Young University

R. McMurray
Brigham Young University

S. McQuarrie
Brigham Young University

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

M. Proudfoot
Brigham Young University

D. Reber
Brigham Young University

C. Robertson
Brigham Young University

R. Sandberg
Brigham Young University

J. Stewart
Brigham Young University

S. Suggs
Brigham Young University

T. Thompson
Brigham Young University

M. Victors
Brigham Young University

J. Webb
Brigham Young University

R. Webb
Brigham Young University

J. West
Brigham Young University

A. Zaitzeff
Brigham Young University

Preface

This lab manual is designed to accompany the textbook *Foundations of Applied Mathematics* by Humpherys and Jarvis.

Contents

Part I

Labs

1 Unix Shell 1: Introduction

Lab Objective: *Explore the basics of the Unix Shell. Understand how to navigate and manipulate file directories. Introduce the Vim text editor for easy writing and editing of text or other similar documents.*

Unix was first developed by AT&T Bell Labs in the 1970s. In the 1990s, Unix became the foundation of Linux and MacOSX. The Unix shell is an interface for executing commands to the operating system. The majority of servers are Linux based, so having a knowledge of Unix shell commands allows us to interact with these servers.

As you get into Unix, you will find it is easy to learn but difficult to master. We will build a foundation of simple file system management and a basic introduction to the Vim text editor. We will address some of the basics in detail and also include lists of commands that interested learners are encouraged to research further.

> **Note**
>
> Windows is not built off of Unix, but it does come with a command line tool. We will not cover the equivalent commands in Windows command line, but you could download a Unix-based shell such as Git Bash or Cygwin to complete this lab (you will still lose out on certain commands).

File System

> **Achtung!**
>
> In this lab you will work with files on your computer. Be careful as you go through each problem and as you experiment on your own. Be sure you are in the right directories and subfolders before you start creating and deleting files; some actions are irreversible.

Navigation

Typically you have probably navigated your comptuer by clicking on icons to open directories and programs. In the terminal, instead of point and click we use typed commands to move from directory to directory.

Begin by opening the Terminal. The text you see in the upper left of the Terminal is called the *prompt*. As you navigate through the file system you will want to know *where* you are so that you know you aren't creating or deleting files in the wrong locations.

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 as the name suggests it prints out the string of your current location.

Once you know where you are, you'll want to know where you can move. The `ls`, or **l**ist **s**egments, command will list all the files and directories in your current folder location. Try typing it in.

When you know what's around you, you'll want to navigate directories. The `cd`, or **c**hange **d**irectory, command allows you to move through 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. Using these commands, navigate to the `Shell1/` directory provided with this lab. We will use this directory for the remainder of the lab. Use the `ls` command to list the contents of this directory. NOTE: You will find a directory within this directory called `Test/` that is availabe for you to experiment with the concepts and commands found in this lab. The other files and directories are necessary for the exercises we will be doing, so take care not to modify them.

Getting Help

As you go through this lab, you will come across many commands with functionality beyond what is taught here. The Terminal has two nice commands to help you with these commands. The first is `man <command>`, which opens the manual page for the command following `man`. Try typing in `man ls`; you will see a list of the name and description of the `ls` command, among other things. If you forget how to use a command the manual page is the first place you should check to remember.

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	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.

Flags

When you typed in `man ls` up above, you may have noticed several options listed in the description, such as `-a`, `-A`, `--author`. These are called flags and change the functionality of commands. Most commands will have flags that change their behavior. Table 1.1 contains some of the most common flags for the `ls` command.

Multiple flags can be combined as one flag. For example, if we wanted to list all the files in a directory in long format sorted by date modified, we would use `ls -a -l -t` or `ls -alt`.

Manipulating Files and Directories

In this section we will learn how to create, copy, move, and delete files and folders. Before you begin, `cd` into the `Test/` directory in `Shell1/`.

To create a text file, use `touch <filename>`. To create a new directory, use `mkdir <dir_name>`.

To copy a file into a directory, use `cp <filename> <dir_name>`. When making a copy of a directory, the command is similar but must use the `-r` flag. This flag stands for recursively copying files in subdirectories. If you try to copy a file 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.

When deleting files, use `rm <filename>`, or `rm -r <dir_name>` when deleting a directory. Again, 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 have the Terminal print strings of what it is doing. When your Terminal gets too cluttered, use `clear` to clean it up.

Below is an example of all these commands in action.

```
$ cd Test
$ touch data.txt                # create new empty file data.txt
$ mkdir New                     # create directory New
$ ls                            # list items in test directory
New     data.txt
$ cp data.txt New/              # copy data.txt to New directory
$ cd New/                       # enter the New directory
$ ls                            # list items in New directory
```

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: The commands discussed in this section.

```
data.txt
$ mv data.txt new_data.txt      # rename data.txt new_data.txt
$ ls                            # list items in New directory
new_data.txt
$ cd ..                         # Return to test directory
$ rm -rv New/                   # Remove New directory and its contents
removed 'New/data.txt'
removed directory: 'New/'
$ clear                         # Clear terminal screen
```

Table 1.2 contains all the commands we have discussed so far. Notice the common flags are contained in square brackets; use `man` to see what these mean.

Problem 2. Inside the `Shell1/` directory, delete the `Audio/` folder along with all its contents. Create `Documents/`, `Photos/`, and `Python/` directories.

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, if you needed 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
```

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.

Command	Description
`cat`	Print the contents of a file in its entirety
`more`	Print the contents of a file one page at a time
`less`	Like more, but you can navigate forward and backward
`head`	Print the first 10 lines of a file
`head -nK`	Print the first `K` lines of a file
`tail`	Print just the last 10 lines of a file
`tail -nK`	Print the last `K` lines of a file

Table 1.4: Commands for printing contents of a file

See Table 1.3 for examples of common wildcard usuage.

Problem 3. Within the `Shell1/` directory, there are many files. We will organize these files into directories. Using wildcards, move all the `.jpg` files to the `Photos/` directory, all the `.txt` files to the `Documents/` directory, and all the `.py` files to the `Python/` directory. You will see a few other folders in the `Shell1/` directory. Do not move any of the files within these folders at this point.

Displaying File Contents

When using the file system, you may be interested in checking file content to be sure you're looking at the right file. Several commands are made available for ease in reading file content.

The `cat` command, followed by the filename will display all the contents of a file on the screen. If you are dealing with a large file, you may only want to view a certain number of lines at a time. Use `less <filename>` to restrict the number of lines that show up at a time. Use the arrow keys to navigate up and down. Press **q** to exit.

For other similar commands, look at table 1.4.

Searching the File System

There are two commands we use for searching through our directories. The `find` command is used to find files or directories in a directory hierarchy. The `grep` command is used to find lines matching a string. More specifically, we can use `grep` to find words inside files. We will provide a basic template in Table 1.5 for using these two commands and leave it to you to explore the uses of the other flags. The `man` command can help you learn about them.

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

Table 1.5: Commands using `find` and `grep`.

Problem 4. In addition to the `.jpg` files you have already moved into the `Photot/` folder, there are a few other `.jpg` files in a few other folders within the `Shell1/` directory. Find where these files are using the `find` command and move them to the `Photos/` folder.

Pipes and Redirects

Terminal commands can be combined using *pipes*. When combined, or *piped*, the output of one command is passed to the another. Two commands are piped together using the `|` operator. To demonstrate pipes we will first introduce commands that allow us to view the contents of a file in Table 1.4.

In the first example below, the `cat` command output is piped to `wc -l`. The `wc` command stands for **w**ord **c**ount. This command can be used to count words or lines. The `-l` flag tells the `wc` command to count lines. Therefore, this first example counts the number of lines in `assignments.txt`. In the second example below, the command lists the files in the current directory sorted by size in descending order. For details on what the flags in this command do, consult `man sort`.

```
$ cd Shell1/Files/Feb
$ cat assignments.txt | wc -l
9

$ ls -s | sort -nr
12 project3.py
12 project2.py
12 assignments.txt
 4 pics
total 40
```

In the previous example, we pipe the contents of `assignments.txt` to `wc -l` using `cat`. When working with files specifically, you can also use *redirects*. The `<` operator gives a file to a Terminal command. The same output from the first example above can be achieved by running the following command:

```
$ wc -l < assignments.txt
9
```

If you are wanting to save the resulting output of a command to a file, 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. For example, if we want to append the number of lines in `assignments.txt` to `word_count.txt`, we would run the following commmand:

```
$ wc -l < assignements.txt >> word_count.txt
```

Since `grep` is used to print lines matching a pattern, it is also very useful to use in conjunction with piping. For example, `ls -l | grep root` prints all files associated with the root user.

Problem 5. The `words.txt` file in the `Documents/` directory contains a list of words that are not in alphabetical order. Write the number of words in `words.txt` and an alphabetically sorted list of words to `sortedwords.txt` using pipes and redirects. Save this file in the `Documents/` directory. Try to accomplish this with a total of two commands or fewer.

Archiving and Compression

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

The `ZIP` file format is the most popular 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`.

```
$ cd Shell1/Documents
$ zip zipfile.zip doc?.txt
 adding: doc1.txt (deflated 87%)
 adding: doc2.txt (deflated 90%)
 adding: doc3.txt (deflated 85%)
 adding: doc4.txt (deflated 97%)

# use -l to view contents of zip file
$ unzip -l zipfile.zip
Archive:  zipfile.zip
  Length     Date     Time    Name
---------  ---------- -----   ----
     5234  2015-08-26 21:21   doc1.txt
     7213  2015-08-26 21:21   doc2.txt
     3634  2015-08-26 21:21   doc3.txt
     4516  2015-08-26 21:21   doc4.txt
---------                     -------
    16081                     3 files

$ unzip zipfile.zip
```

```
  inflating: doc1.txt
  inflating: doc2.txt
  inflating: doc3.txt
  inflating: doc4.txt
```

While the zip file format is more popular on the Windows platform, the `tar` utility is more common in the Unix environment. The following commands use `tar` to archive the files and `gzip` to compress the archive.

Notice that all the commands below have the `-z`, `-v`, and `-f` flags. The `-z` flag calls for the `gzip` compression tool, the `-v` flag calls for a verbose output, and `-f` indicates the next parameter will be the name of the archive file.

```
$ ls
doc1.txt    doc2.txt    doc3.txt    doc4.txt

# use -c to create a new archive
$ tar -zcvf docs.tar.gz doc?.txt
doc1.txt
doc2.txt
doc3.txt
doc4.txt

$ ls
docs.tar.gz

# use -t to view contents
$ tar -ztvf <archive>
-rw-rw-r-- username/groupname 5119 2015-08-26 16:50 doc1.txt
-rw-rw-r-- username/groupname 7253 2015-08-26 16:50 doc2.txt
-rw-rw-r-- username/groupname 3524 2015-08-26 16:50 doc3.txt
-rw-rw-r-- username/groupname 4516 2015-08-26 16:50 doc4.txt

# use -x to extract
$ tar -zxvf <archive>
doc1.txt
doc2.txt
doc3.txt
doc4.txt
```

Problem 6. Archive and compress the files in the `Photos/` directory using `tar` and `gzip`. Name the arhive `pics.tar.gz` and save it inside the `Photos/` directory. Use `ls -l` to see how much the files were compressed in the process.

Vim: A Terminal Text Editor

Today many have become accustomed to having GUIs (Graphic User Interfaces) for all their applications. Before modern text editors (i.e. Microsoft Word, Pages for Mac, Google Docs) there were terminal text editors. Vim is one of the most popular terminal text editors. While vim may be intimidating at first, as you become familiar with vim it may become one of your preferred text editors for writing code.

One of the major philosophies of vim is to be able to keep your fingers on the keyboard at all times. Thus, vim has many keyboard shortcuts that allow you to navigate the file and execute commands without relying on a mouse, toolbars, or arrow keys.

In this section, we will go over the basics of navigation and a few of the most common commands. We will also provide a list of commands that interested readers are encouraged to research.

It has been said that at no point does somebody finish learning Vim. You will find that you will constantly be able to add something new to your arsenal.

Getting Started

Start Vim with the following command:

```
$ vim my_file.txt
```

When executing this command, if `my_file.txt` already exists, vim will open the file and we may begin editing the existing file. If `my_file.txt` does not exist, it will be created and we may begin editing the file.

You may notice if you start typing the characters may or may not appear on your screen. This is because vim has multiple modes. When vim starts, we are placed in *command mode*. We want to be in *insert mode* to begin entering text. To enter insert mode from command mode, hit the `i` key. You should see `-- INSERT --` at the bottom of your terminal window. In insert mode vim act like a typical word processor. Letters will appear in the document as you type them. If you ever need to leave insert mode and return to command mode, hit the `Esc` key.

Saving/Quitting Vim

To save or quit the current document, first enter last line mode by pressing the **:** key. To just save, type **w** and hit enter. To save and quit, type **wq**. To quit without saving, run **q!**

Problem 7. Using vim, create a new file in the `Documents/` directory named `first_vim.txt`. Write least multiple lines to this file. Save and exit the file you have created.

Navigation

We are accustomed to navigating GUI text editors using a mouse and arrow keys. In vim, we navigate using keyboard shortcuts while in command mode.

Command	Description
`a`	**a**ppend text after cursor
`A`	**A**ppend text to end of line
`o`	Begin a new line below the cursor
`O`	Begin a new line above the cursor
`s`	Substitute characters under cursor

Table 1.6: Commands for entering insert mode

Problem 8. Become accustomed to navigating in command mode using the following keys:

Command	Description
`k`	up
`j`	down
`h`	left
`l`	right
`w`	beginning of next **w**ord
`e`	**e**nd of next word
`b`	**b**eginning of previous word
`0`	(zero) beginning of line
`$`	end of line
`gg`	beginning of file
`#gg`	go to line `#`
`G`	end of file

Alternative Ways to Enter Insert Mode

Hitting the `i` key is not the only way to enter insert mode. Alternative methods are described in Table 1.6.

Visual Mode

Visual mode allows you to select multiple characters. Among other things, we can use this to replace words with the `s` command, and we can select text to cut or copy.

Problem 9. Open the document you created in the previous problem. While in command mode, enter visual mode by pressing the `v` key. Using the navigation keys discussed earlier, move the cursor to select a few words. Copy this text using the `y` key (stands for **y**ank). Return to command mode by pressing `Esc`. Move the cursor to where you would like to paste the text and press the `p` key to paste. Similarly, select text in visual mode and hit `d` to **d**elete the text and paste it somewhere else with the `p` key.

Command	Description
`x`	delete letter after cursor
`X`	delete letter before cursor
`dd`	delete line
`dl`	**d**elete **l**etter
`d#l`	**d**elete # **l**etters
`dw`	**d**elete **w**ord
`d#w`	**d**elete # **w**ords

Table 1.7: Commands for deleting in command mode

Command	Description
`:map`	customize
`:help`	view vim docs
`cw`	**c**hange **w**ord
`u`	undo
`Ctrl-R`	redo
`.`	Repeat the previous command
`*`	find next occurrence of word under cursor
`#`	find previous occurrence of word under cursor
`/str`	find `str` in file
`n`	find next match
`N`	find previous match

Table 1.8: Commands for entering insert mode

Deleting Text in Command Mode

Insert mode should only be used for inserting text. Try to get in the habit of leaving insert mode as soon as you are done adding the text you want to add. Deleting text is much more efficient and versatile in command mode. The `x` and `X` commands are used to delete single characters. The `d` command is always accompanied by another navigational command. See Table 1.7 for a few examples.

A Few Closing Remarks

In the next lab, we will introduce how to access another machine through the terminal. Vim will be essential in this situation since GUIs will not be an option.

If you are interested in continuing to use vim, you may be interested in checking out *gvim*. Gvim is a GUI that uses vim commands in a more traditional text editor window.

Also, in Table 1.8, we have listed a few more commands that are worth exploring. If you are interested in any of these features of vim, we encourage you to research these features further on the internet. Additionally, many people have published their `vimrc` file on the internet so other vim users can learn what options are worth exploring. It is also worth noting that we can use vim navigation commands in many other places in the shell. For example, try using the navigation commands when viewing the `man vim` page.

2 More on the Unix Shell

Lab Objective: *Introduce system management, calling Unix Shell commands within Python, and other advanced topics.*

In this lab, we will build upon the foundation of the previous lab. As in the last lab, the majority of learning will not be had in finsihing the problems, but in following the examples. By the end of this lab, you will have a solid foundation in Unix. You will be able to understand enough to learn any additional topics you want.

File Security

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

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

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

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

Table 2.1: Symbolic permissions notation

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

Table 2.2: Octal permissions notation

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

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

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

Scripts

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

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

Problem 1. Using vim, create a file called `hello` that contains the previous text and save it. Note that no file type is necessary.

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

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

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

# Notice you do not have permission to execute this file. This is by default.
$ ls -l hello
```

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

Table 2.3: Other commands when working with permissions

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

Table 2.4: Commands for resource management

```
-rw-r--r-- 1 username groupname 31 Jul 30 14:34 hello
```

Problem 2. Add executable permission to your `hello` script. Run the script again.

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

Problem 3. In the `Python/` directory you will find `count_files.py`. `count_files.py` is a python script that counts all the files within the `Shell2/` directory. Modify this file so it can be run as a script and change the permissions of this script so the user and group can execute the script.

Note: In the `subprocess.check_output` command, if `Shell2/` is not contained in your home directory (∼), you will need to change ∼ to the correct path to navigate there.

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

Resource Management

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

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

Table 2.5: Job control commands

Job Control

Let's say we had a series of scripts we wanted to run. If we knew that these would take a while to execute, we may want to start them all at the same time and let them run while we are working on something else. In table 2.5, we have listed some of the most common commands used in job control. We strongly encourage you to experiment with these commands. In the `Scripts/` directory, you will find a `five_secs` and a `ten_secs` script that takes five seconds and ten seconds to execute respectively. These will be particularly useful as you are experimenting with these commands.

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

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

Python Integration

To this point, we have barely scratched the surface of all the functionality that Unix has to offer. However, the tools and commands we have addressed so far provide us with a foundation of the basics. Using the `subprocess` module in Python, we can call Unix commands. By combining Python and the Unix commands, our toolset is automatically broadened.

There are two functions in particular within the `subprocess` module we will use. When wanting to run a Unix command, use `subprocess.call()`. When wanting to run a Unix command and be able to store and manipulate the output, use `subprocess.check_output()`. These functions have a keyword argument `shell` that defaults to `False`. We want to set this argument to `True` to run the command in the Unix shell.

```
$ cd Shell-Lab/Documents
$ python
>>> import subprocess
>>> subprocess.call("ls -l", shell=True)
-rw-rw-r-- 1 username groupname  142 Aug  5 20:20 assignments.txt
-rw-rw-r-- 1 username groupname  427 Aug  5 20:21 doc1.txt
-rw-rw-r-- 1 username groupname  326 Aug  5 20:21 doc2.txt
-rw-rw-r-- 1 username groupname  612 Aug  5 20:21 doc3.txt
-rw-rw-r-- 1 username groupname  298 Aug  5 20:21 doc4.txt
-rw-rw-r-- 1 username groupname 1027 Aug  5 20:23 review.txt
-rw-rw-r-- 1 username groupname  920 Aug  5 23:50 words.txt
>>> files = subprocess.check_output("ls -l", shell=True)
>>> files
'-rw-rw-r-- 1 username groupname  142 Aug  5 20:20 assignments.txt\n-rw-rw-r-- ↩
    1 username groupname  427 Aug  5 20:21 doc1.txt\n-rw-rw-r-- 1 username ↩
    groupname  326 Aug  5 20:21 doc2.txt\n-rw-rw-r-- 1 username groupname  612 ↩
    Aug  5 20:21 doc3.txt\n-rw-rw-r-- 1 username groupname  298 Aug  5 20:21 ↩
    doc4.txt\n-rw-rw-r-- 1 username groupname 1027 Aug  5 20:23 review.txt\n-rw↩
    -rw-r-- 1 username groupname  920 Aug  5 23:50 words.txt\n'
>>> files.split('\n')
['-rw-rw-r-- 1 username groupname  142 Aug  5 20:20 assignments.txt',
 '-rw-rw-r-- 1 username groupname  427 Aug  5 20:21 doc1.txt',
 '-rw-rw-r-- 1 username groupname  326 Aug  5 20:21 doc2.txt',
 '-rw-rw-r-- 1 username groupname  612 Aug  5 20:21 doc3.txt',
 '-rw-rw-r-- 1 username groupname  298 Aug  5 20:21 doc4.txt',
 '-rw-rw-r-- 1 username groupname 1027 Aug  5 20:23 review.txt',
 '-rw-rw-r-- 1 username groupname  920 Aug  5 23:50 words.txt',
 '']
>>> files = files.split('\n')
# To get rid of the last empty string in the list
>>> files.pop()
''

# Now that we have a list object, we can manipulate and analyze this data in ↩
    Python.
We can make it even more accessible by splitting the lines again
>>> files = [line.split() for line in files]
```

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.

Problem 5. Create a `Shell` class in Python. Write a `find_file()` method that will search for a filename using the `find` command in the given directory. Write a `find_word()` method that finds a given word within the contents of the directory using the `grep` command. Both functions should accept a directory keyword as input which defaults to `None`. If no directory location is provided, then set it to be the current directory within the function. For both these functions, return a list of filepaths.

Problem 6. Write a method for the `Shell` class that recursively finds the n largest files within a directory. Have a keyword argument for the directory that defaults to the current directory. Be sure that your function only returns files. Hint: To view the size of a file `file1`, you can use `ls -s file1` or `du file1`

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.

Secure Shell

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

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

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

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

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

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

$ hostname
my_hostname

$ exit
logout
Connection to my_host_name closed.
```

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

Secure Copy

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

```
# copy filename to the host's system at filepath
$ scp filename host_username@hostname:filepath

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

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

Problem 7. You will either need a partner for this problem or have access to a username on another computer. Experiment with SSH. Verify that you can connect from a client to a host. Copy a few files between the host and the client.

Generating SSH Keys (Optional)

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

Web Related

Sometimes you will need to download files from the internet. `wget` and `curl` are both used to download content from the web, and in many applications they both perfform the same tasks. Most of the differences between `wget` and `curl` are beyond the scope of this book. At its most basic, `curl` is the more robust tools of the two while `wget` can download recursively. The provided examples will use `wget`.

Downloading files using Wget

When we want to download a single file, we just need the URL for the file we want to download. Running the command below will downlad a JPEG image of a person writing on a chalkboard. Similarly, you can download PDF files, HTML files, and other content simply by providing a different URL.

```
$ wget http://acme.byu.edu/wp-content/uploads/2013/07/0906-13-00903.jpg
```

The following are also useful commands using `wget`.

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

# Download in the background
$ wget -b URL

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

Problem 8. In the `Documents/` directory, you will find a file named `urls.txt` with a list of URLs. Download the files in this list using `wget`. Move the pictures that will be downloaded to the `Photos/` directory.

Additional Material

sed and awk

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

Printing Specific Lines Using sed

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

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

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

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

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

Find and Replace Using sed

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

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

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

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

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

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

Formatting output using awk

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

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

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

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

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

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

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

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

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

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

Problem 9. 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 a file 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 this can be accomplished using one 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`.

3 Basic Regular Expressions

Lab Objective: *Learn the basics of using regular expressions to find text*

Regular expressions allow for quick searching and replacing of general patterns of text. While nearly all text editors have a feature that will find and replace exact strings of text, regular expressions are used to find text in a much more general way. For example, using a single regular expression, you can find every email address in a text file without having to sift through it by hand.

Terminology and Basics

A "regular expression" is basically just a string of characters that follow a certain syntax. Computer programs can then interpret these expressions as instructions to search for certain kinds of text. We will often call regular expressions "patterns", and we will say that certain patterns "match" certain strings. The general idea is that a regular expression represents a large set of strings (for example, all valid email addresses), and if a specific string is in that set, we say that the regular expression matches that string.

> ACHTUNG!
>
> Regular expression libraries have been implemented and are a part of the standard distribution of nearly every programming language, and many text editors have a find-and-replace mode that uses regular expressions. Unfortunately, the syntax for regular expressions may be slightly different in each implementation. There is no universal standard for all regular expressions across all platforms. However, the orginal syntax and a few variants are very widespread, so the basic regular expression techniques we learn in this lab should be virtually the same in almost every situation you will encounter them.

The simplest use of regular expressions is to match text literally. For example, the pattern `"cat"` matches the string `"cat"` but does not match the strings `"dog"` or `"bat"`.

Now that we have a general idea of what regular expressions are, we will see how to use them in Python.

Regular Expressions in Python

The python package `re` contains the functionality for using regular expressions. To use it, simply run the command `import re`.

The following Python code demonstrates what we said earlier about the regular expression `"cat"`:

```
>>> bool(re.match("cat", "cat"))
True
>>> bool(re.match("cat", "dog"))
False
>>> bool(re.match("cat", "bat"))
False
```

The main functions we will use are `re.match(pattern, string_to_test)` and `re.compile(pattern)`. You can think of `re.match` as returning a boolean value representing whether the given pattern matched the given string. The function `re.compile` returns a compiled object that represents a regular expression. You can then call the `match` function on this compiled object to get a boolean value. There is a similar function, `re.search`, which will match the regular expression anywhere inside a given string. We will see one example shortly where `re.search` is preferred in multiline matching.

The following code shows an example of how to use `re.compile`:

```
>>> pattern = re.compile("any regular expression")
>>> result = pattern.match("any string")
```

The above code is equivalent to the following:

```
>>> result = re.match("any regular expression", "any string")
```

Most programs use the compiled form (the first of the above two examples) for efficiency.

When constructing a regular expression, it is best to construct your pattern string using Python's syntax for raw strings by prefacing the string with the `'r'` character. This causes the constructed string to treat backslashes as actual backslash characters, rather than the start of an escape sequence.

For example:

```
>>> normal = "hello\nworld"
>>> raw = r"hello\nworld"
>>> print normal
hello
world
>>> print raw
hello\nworld
>>> type(normal), normal
(str, 'hello\nworld')
>>> type(raw), raw
(str, 'hello\\nworld')
```

Note that `raw` and `normal` are both python strings; one was just constructed differently. Also notice that when we constructed `raw`, it inserted an extra backslash before the existing backslash.

We use raw strings because the backslash character is a very important special character in regular expressions. If we wanted to use backslash characters as part of a normally-constructed Python string, we would need to either escape every single backslash by using two backslashes each time, or we could take the much easier and less confusing route of using Python's raw strings. To demonstrate this effect, suppose we wanted to know whether the regular expression `"\$3\.00"` matched the string `"$3.00"`. We could get our answer in either of the following ways:

```
>>> bool(re.match("\\$3\\.00", "$3.00"))
True
>>> bool(re.match(r"\$3\.00", "$3.00"))
True
```

(You will see why this pattern matches this string soon)

Remember, readability counts.

Literal Characters and Metacharacters

The following characters are used as metacharacters in regular expressions:

```
. ^ $ * + ? { } [ ] \ | ( )
```

These characters mean special things when used in regular expressions, making the vast power of regular expressions possible. We will get to using these characters later. For now, what do we do if want to match these characters literally? We simply escape these characters using the metacharacter `'\'`:

```
>>> pattern = re.compile(r"\$2\.95, please")
>>> bool(pattern.match("$2.95, please"))
True
>>> bool(pattern.match("$295, please"))
False
>>> bool(pattern.match("$2.95"))
False
```

Problem 1. Define the variable `pattern_string` using literal characters and escaped metacharacters in such a way that the following python program prints `True`:

```
import re
pattern_string = r""  # Edit this line
pattern = re.compile(pattern_string)
print bool(pattern.match("^{(!%.*_)}&"))
```

A little misleadingly, the `re.match` method isn't actually checking whether the given regular expression matches entire strings. Rather, it checks whether the regular expression matches *at* the *beginning* of the string, even if the string continues on afterward. For example:

```
>>> pattern = re.compile(r"x")
>>> bool(pattern.match("x"))
True
>>> bool(pattern.match("xabc"))
True
>>> bool(pattern.match("abcx"))
False
```

You might not expect the pattern `'x'` to match the string `"xabc"`, but it does. This can cause confusion and headache, so we'll have to be a little more precise with the help of metacharacters.

The *line anchor* metacharacters, `'^'` and `'$'`, are used to match the start and the end of a line of text, respectively. Let's see them in action:

```
>>> pattern = re.compile(r"^x$")
>>> bool(pattern.match("x"))
True
>>> bool(pattern.match("xabc"))
False
>>> bool(pattern.match("abcx"))
False
```

An added 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. Since we have seen two ways to match strings with regex expressions, the following shows two ways to implement multiline searching:

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

For simplicity, the rest of the lab will focus on single line matching.

Let's move on to `'('`, `')'`, and `'|'`. The `'|'` character (the "pipe" character, usually found on the key below the backspace key) matches one of two or more regular expressions:

```
>>> pattern2 = re.compile(r"^red$|^blue$")
>>> pattern3 = re.compile(r"^red$|^blue$|^orange$")
>>> bool(pattern2.match("red")),     bool(pattern3.match("red"))
```

```
(True, True)
>>> bool(pattern2.match("blue")),    bool(pattern3.match("blue"))
(True, True)
>>> bool(pattern2.match("orange")),  bool(pattern3.match("orange"))
(False, True)
>>> bool(pattern2.match("redblue")), bool(pattern3.match("redblue"))
(False, False)
```

You can think of `'|'` as doing an "or" operation. How would we create a regular expression that matched both `"one fish"` and `"two fish"`? Although the regular expression `"one fish|two fish"` works, there is a better way, by using both the pipe character and parentheses:

```
>>> pattern = re.compile(r"^(one|two) fish$")
>>> bool(pattern.match("one fish"))
True
>>> bool(pattern.match("two fish"))
True
>>> bool(pattern.match("three fish"))
False
>>> bool(pattern.match("one two fish"))
False
```

As the above example demonstrates, parentheses are used to group sequences of characters together and change the order of precedence of the metacharacters, much like how parentheses work in an arithmetic expression such as `3*(4+5)`. In regular expressions, the `'|'` metacharacter has the lowest precedence out of all the metacharacters.

Parentheses actually have more uses, which we will learn later. For now, note that parentheses aren't matched literally:

```
>>> bool(re.match(r"r(hi)no(c(e)ro)s", "rhinoceros"))
True
```

Parentheses help give regular expressions higher precedence. For example, `"^one|two fish$"` gives precedence to the invisible string concatenation between `"two"` and `"fish"` while `"^(one|two) fish$"` gives precedence to the `'|'` metacharacter.

Problem 2. Define the variable `pattern_string` using the metacharacter `'|'` and parentheses in such a way that the following python program prints `True`:

```
import re
pattern_string = r""  # Edit this line
pattern = re.compile(pattern_string)
strings_to_match = [ "Book store", "Book supplier", "Mattress store", "↩
    Mattress supplier", "Grocery store", "Grocery supplier"]
print all(pattern.match(string) for string in strings_to_match)
```

Your regular expression should not match any other string, including strings such as `"Book store sale"`.

Character Classes

The metacharacters `'['` and `']'` are used to create *character classes*. Here they are in action:

```
>>> pattern = re.compile(r"[xy]")
>>> bool(pattern.match("x"))
True
>>> bool(pattern.match("y"))
True
>>> bool(pattern.match("z"))
False
>>> bool(pattern.match("x: Why does this match? Were you paying attention?"))
True
```

In essence, a character class will match any one out of several characters.

Inside character classes, there are two additional metacharacters: `'-'` and `'^'`. Although we've already seen `'^'` as a metacharacter, it has a different meaning when used inside a character class. When `'^'` appears *as the first character* in a character class, the character class matches anything not specified instead. Think of `'^'` as performing a set complement operation on the character class. For example:

```
>>> pattern = re.compile(r"^[^ab]$")
>>> bool(pattern.match("x"))
True
>>> bool(pattern.match("#"))
True
>>> bool(pattern.match("a"))
False
>>> bool(pattern.match("b"))
False
```

Note that the two `'^'` characters mean completely different things; the first `'^'` anchors us at the beginning of the line, while the second `'^'` performs a set complement operation on the character class `"[ab]"`.

The other character class metacharacter is `'-'`. This is used to specifiy a range of values. For example:

```
>>> pattern = re.compile(r"^[a-z][0-9][0-9]$")
>>> bool(pattern.match("a90"))
True
>>> bool(pattern.match("z73"))
True
>>> bool(pattern.match("A90"))
False
>>> bool(pattern.match("zs3"))
```

```
False
```

Multiple ranges or characters can be included in a single character class; in this case, the character class will match any character that fits either criterion:

```
>>> pattern = re.compile(r"^[abcA-C][0-27-9]$")
>>> bool(pattern.match("b8"))
True
>>> bool(pattern.match("B2"))
True
>>> bool(pattern.match("a9"))
True
>>> bool(pattern.match("a4"))
False
>>> bool(pattern.match("E1"))
False
```

Notice in the first line that `[abcA-C]` acts like `[a|b|c|(A-C)]` and `[0-27-9]` acts like `[(0-2)|(7-9)]`.

Finally, there are some built-in shorthands for certain character classes:

- `'\d'` (think "digit") matches any digit. It is equivalent to `"[0-9]"`.
- `'\w'` (think "word") matches any alphanumeric character or underscore. It is equivalent to `"[a-zA-Z0-9_]"`.
- `'\s'` (think "space") matches any whitespace character. It is equivalent to `"[ \t\n\r\f\v]"`.

The following character classes are the complements of those above:

- `'\D'` is equivalent to `"[^0-9]"` or `"[^D]"`
- `'\W'` is equivalent to `"[^a-zA-Z0-9_]"` or `"[^W]"`
- `'\S'` is equivalent to `"[^ \t\n\r\f\v]"` or `"[^S]"`

These character classes can be used in character classes; for example, `"[_A-Z\s]"` will match an underscore, any capital letter, or any whitespace character.

The `'.'` metacharacter, equivalent to `"[^\n]"` on UNIX and `"[^\r\n]"` on Windows, matches any character except for a line break. For example:

```
>>> pattern = re.compile(r"^.\d.$")
>>> bool(pattern.match("a0b"))
True
>>> bool(pattern.match("888"))
True
>>> bool(pattern.match("n2%"))
True
>>> bool(pattern.match("abc"))
False
>>> bool(pattern.match("m&m"))
```

```
False
>>> bool(pattern.match("cat"))
False
```

Problem 3. Define the variable `pattern_string` in such a way that the following python program prints `True`:

```
import re
pattern_string = r""  # Edit this line
pattern = re.compile(pattern_string)

strings_to_match = ["a", "b", "c", "x", "y", "z"]
uses_line_anchors = (pattern_string.startswith('^') and pattern_string.endswith('$'))
solution_is_clever = (len(pattern_string) == 8)
matches_list = all(pattern.match(string) for string in strings_to_match)

print uses_line_anchors and solution_is_clever and matches_list
```

Problem 4. A *valid python identifier* (aka a valid variable name) is defined as any string composed of an alphabetic character or underscore followed by any (possibly empty) sequence of alphanumeric characters and underscores.

Define the variable `identifier_pattern_string` that defines a regular expression that matches valid python identifiers that are exactly five characters long.

To help you test your pattern, the following program should print `True`. (This is necessary but not sufficient to show your regular expression is correct):

```
import re
identifier_pattern_string = r""  # Edit this line
identifier_pattern = re.compile(identifier_pattern_string)

valid = ["mouse", "HORSE", "_1234", "__x__", "while"]
not_valid = ["3rats", "err*r", "sq(x)", "too_long"]

print all(identifier_pattern.match(string) for string in valid) and not any(identifier_pattern.match(string) for string in not_valid)
```

Hint: Use the `'\w'` character class to keep your regular expression relatively short.

> **Note**
>
> As you might have noticed, using this definition, `"while"` is considered a valid python identifier, even though it really is a reserved word. In the following problems, we will make a few other simplifying assumptions about the python language.

Repetition

Suppose in the last problem we wanted the string to be 20 characters long. You wouldn't want to write `\w` 20 times. In fact, what if you wanted to match at most one instance of a character or a number with at least three digits? The metacharacters `'*'`, `'+'`, `'{'`, and `'}'` are very useful for repetition.

The `'*'` metacharacter means "Match zero or more times (as many as possible)" when it follows another regular expression. For instance:

```
>>> pattern = re.compile(r"^a*b$")
>>> bool(pattern.match("b"))
True
>>> bool(pattern.match("ab"))
True
>>> bool(pattern.match("aab"))
True
>>> bool(pattern.match("aaab"))
True
>>> bool(pattern.match("abab"))
False
>>> bool(pattern.match("abc"))
False
```

The `'+'` metacharacter means "Match one or more times (as many as possible)" when it follows another regular expression. As an example:

```
>>> pattern = re.compile(r"^h[ia]+$")
>>> bool(pattern.match("ha"))
True
>>> bool(pattern.match("hii"))
True
>>> bool(pattern.match("hiaiaa"))
True
>>> bool(pattern.match("h"))
False
>>> bool(pattern.match("hah"))
False
```

It's important to understand why `"hiaiaa"` *is* a match here; matching multiple times means matching the preceeding *expression* multiple times, not matching the *results* of the preceeding expression multiple times. We haven't yet learned how to construct a regular expression with that behavior.

The `'?'` metacharacter means "Match one time (if possible) or do nothing (i.e. match zero times)" when it follows another regular expression:

```
>>> pattern = re.compile(r"^abc?$")
>>> bool(pattern.match("abc"))
True
>>> bool(pattern.match("ab"))
True
>>> bool(pattern.match("abd"))
False
>>> bool(pattern.match("ac"))
False
```

The curly brace metacharacters are used to specify a more precise amount of repetition:

```
>>> pattern = re.compile(r"^a{2,4}$")
>>> bool(pattern.match("a"))
False
>>> bool(pattern.match("aa"))
True
>>> bool(pattern.match("aaa"))
True
>>> bool(pattern.match("aaaa"))
True
>>> bool(pattern.match("aaaaa"))
False
```

If two arguments `x` and `y` are given to the curly braces (i.e., `{x, y}`), the preceeding regular expression must appear between `x` and `y` times, inclusive, in order for the overall expression to match.

Achtung!

In this last example, line anchors can save us from a lot of confusion. Note the differences between the following example and the example immediately above:

```
>>> pattern = re.compile(r"a{2,4}")
>>> bool(pattern.match("a"))
False
>>> bool(pattern.match("aa"))
True
>>> bool(pattern.match("aaa"))
True
>>> bool(pattern.match("aaaa"))
True
```

```
>>> bool(pattern.match("aaaaa"))
True
```

If only one argument `x` is given and is followed by a comma, the preceeding regular expression must match `x` or more times. If only one argument `x` is given without a comma, the preceeding regular expression must match *exactly* `x` times. For example:

```
>>> exactly_three = re.compile(r"^a{3}$")
>>> three_or_more = re.compile(r"^a{3,}$")
>>> def test_both_patterns(string):
...     return bool(exactly_three.match(string)), bool(three_or_more.match(↩
    string))
>>> test_both_patterns("a")
(False, False)
>>> test_both_patterns("aa")
(False, False)
>>> test_both_patterns("aaa")
(True, True)
>>> test_both_patterns("aaaa")
(False, True)
>>> test_both_patterns("aaaaa")
(False, True)
```

You can also test `{,x}` which will match the preceeding regular expression up to `x` times.

Problem 5. Modify your definition of `identifier_pattern_string` from the previous problem to match valid python identifiers of any length.

Cleaning Dirty Data with Regular Expressions

A common consensus among data scientists is that the majority of your time will be spent cleaning data. Throughout the remainder of this volume, you will have multiple opportunities to practice cleaning data.

Often times, cleaning data is as simple as changing the format of your data or filling missing values. However, with text-based data, additional work is often necessary. Using regular expressions to clean text-based data is often a good option.

Problem 6. The provided file `contacts.txt` contains poorly formatted contact data for 5000 (fictitious) individuals. This dataset contains birthdays, email addresses, and phone numbers for the individuals.

You will notice that much of this data is missing. To make things more complicated, the format of the data isn't consistent. For example, some birthdays are in the format `1/1/99`, some in the format `01/01/1999`, and some in the format `1/1/1999`. The formatting for phone numbers is not consistent either. Some phone numbers are of the form `(123)456-7890` while others are of the form `123-456-7890`.

Using regular expressions, create a Python dictionary where the key is the name of the individual and the value is a dictionary of data. For example, the resulting dictionary should look something like this:

```
{"John Doe":{"bday":"1/1/1990",
             "email":"john_doe90@gmail.com",
             "phone":"(123)456-7890"}}
```

Additional Material

Regular Expressions in the Unix Shell

As we have seen thusfar, 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

As in Lab 2, we will be using `awk` to format output. By incorporating regular expressions, `awk` 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."

Problem 7. You have been given a list of transactions from a fictional start-up company. In the `transactions.txt` file, each line represents a transaction. Transactions are represented as follows:

```
# Notice the semicolons delimiting the fields. Also, notice that in ↩
    between the last and first name, that is a comma, not a semicolon.
<ORDER_ID>;<YEAR><MONTH><DAY>;<LAST>,<FIRST>;<ITEM_ID>
```

Using this set of transactions, produce the following information using regular expressions and the given command:

- Using `grep`, print all transactions by either Nicholas Ross or Zoey Ross.
- Using `awk`, print a sorted list of the names of individuls that bought item 3298.
- Using `awk`, print a sorted list of items purchased between June 13 and June 15 of 2014 (inclusive).

These queries can be produced using one command each.

We encourage the interested reader to research more about how regular expressions can be used with `sed`.

4 SQL and Relational Databases

Lab Objective: *Understand concepts of a relational database and the fundamentals of the SQL language via SQLite.*

When working with large amounts of data, it is important to be able to quickly find and retrieve interesting information. Fortunately, there is a way to handle such massive amounts of data in a reasonably efficient way: a database. A database is simply a structured repository of data, and it allows us to store and retrieve information very quickly. It is managed by a *database management system*, or DBMS. The DBMS is software that allows users to interact directly with the database.

Relational Databases

A *relational database* is a paradigm for organizing data inside of a database. In this paradigm, the data is broken down into tuples of information. These tuples are then grouped into tables, or *relations*, each of which is simply a set of tuples. Each table has a *schema* that defines the attributes of the tuples within the table. If we fix an order to the attributes in the schema, we can think of each attribute as a column of the table, and each tuple as a row of the table. See Figure 4.1 for an illustration of these ideas.

As an example, suppose we have demographic data for a large number of individuals. If we are interested in the gender and age of the individuals, we might make a table with schema (Name, Gender, Age). This table would consist of several 3-tuples, such as (Jane Doe, F, 20). Alternatively, we can view this table as having three columns and as many rows as there are individuals within our data set. We might also create a table with schema (Name, Employment Status, Income, Education).

In the relational paradigm, there must be at least one attribute in each schema that can act as a *primary key*. This can uniquely identify each tuple of the table. It is common to use an ID number or other such unique information for the primary key. In our example above, the “Name" attribute acted as a primary key. However, this attribute only works as a primary key provided no two individuals within the data set have the same name.

One important feature of a database is the *transaction*, which is a conceptual protocol for interacting with the database. Most relational databases are transactional databases. The best way to conceptualize this is imagine that your database is like a bank. Your connection to the database is analogous to the bank teller. When you make one or more deposits and withdrawals, you are making a transaction. A database transaction should have certain properties to protect the integrity of the data. These properties are described in detail in `en.wikipedia.org/wiki/ACID`

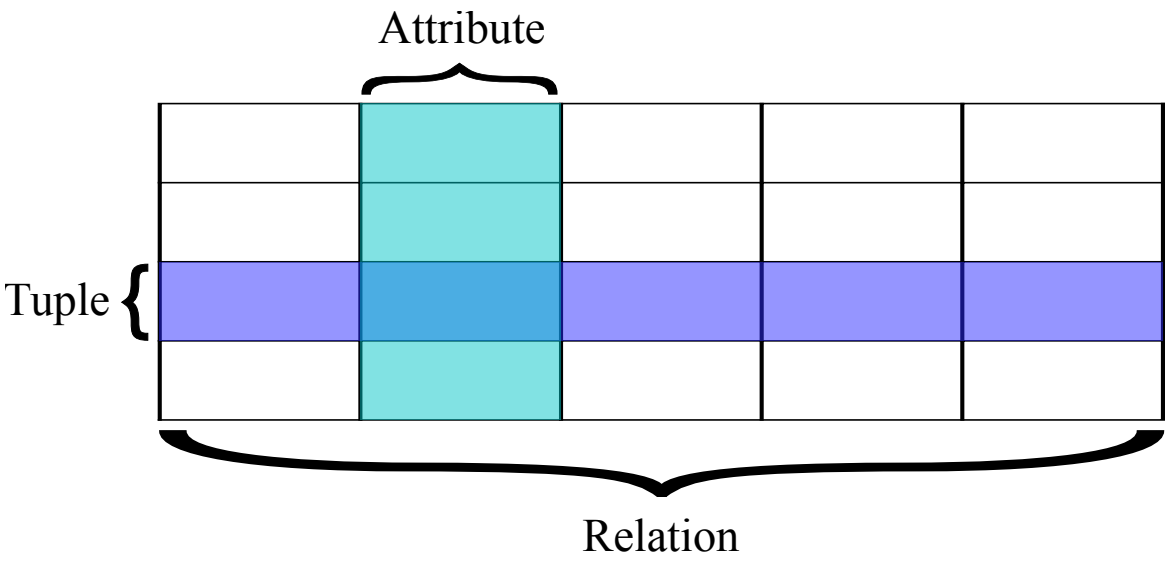


Figure 4.1: Elements of a relation.

Introduction to SQL

Most common DBMSs use a variant of the SQL language to interact with the database. SQL is an acronym for *Structured Query Language*, and may be pronounced like the word “sequel" or by saying the letters “s", “q", and “l" separately. While SQL is not generally portable across different DBMSs, we will focus on the parts of SQL that are relatively common. In particular, we will base our discussion on the SQLite database management system, a very popular DBMS.

SQL consists of blocks of code called statements. Each statement is made up of clauses which may or may not require predicates. Predicates specify conditions that can limit the effect of a clause.

> **Note**
>
> SQL commands are often written in all caps to help distinguish them from the other parts of the query. It is only a matter of style. SQLite, along with most other database managers, is case insensitive. In Python’s SQL interface, the semicolon is also not needed. However, most other database systems will require it, so it’s a good idea to conform in Python.

Let’s look at an example SQL statement:

```
SELECT * FROM table WHERE id=3+1 AND name='Bob';
```

This statement includes a SELECT clause and a WHERE clause. The WHERE clause contains two predicates: `id=3+1` and `name='Bob'`. These two predicates limit the effect of the SELECT clause because any resulting tuples in the table must satisfy both conditions. This entire statement is classified as a query since it does not modify the database in any way.

SQL has several classes of statements. The two main classes we will cover in this lab are schema (Table 4.1) and data manipulation (Table 4.2). We will give you a simplified description of each command and its syntax. You are encouraged to look up the full syntax outside of this lab.

Keyword	Syntax
CREATE TABLE	CREATE TABLE <table> (<col1> <type>, <col2> <type>, ...);
DROP TABLE	DROP TABLE <table>;
CREATE INDEX	CREATE INDEX <name> ON <table> (<col>);
DROP INDEX	DROP INDEX <name>;

Table 4.1: The SQL Schema commands

Keyword	Syntax
INSERT INTO	INSERT INTO <table> <attributes> VALUES (<value1>, <value2>, ...);
UPDATE	UPDATE <table> SET (<col1>=<val1>, <col2>=<val2>, ...) WHERE <condition>;
DELETE	DELETE FROM <table> WHERE <condition>;
SELECT	SELECT <attributes> FROM <table> WHERE <condition>;

Table 4.2: The SQL Data Manipulation commands

SQL in Python

Python has built-in support for SQLite databases using the standard library. Let's open a database called `test1`.

```
import sqlite3 as sql
db = sql.connect("test1")
```

The `connect()` function is used to connect to a database. If it does not already exist, then a new database will be created using the string passed as the argument for the name. The new database was created as a file in the current working directory.

Ending the SQL Session

Once we are finished performing SQL statements and interacting with the database, we need to commit our changes and safely close the connection to the database. This can be done by calling methods on the database connection object.

```
db.commit()     #save changes made in the transaction
db.close()      #safely close the database
```

A database connection is automatically closed in Python when the connection object is garbage-collected. However, it is nice to be safe and explicit in closing a database connection using the `close()` method.

Cursor

To execute SQL commands, we need to get a cursor object from the database.

```
cur = db.cursor()
```

Method	Description
`execute`	Execute a single SQL statement
`executemany`	Execute a single SQL statement over a sequence
`executescript`	Execute a SQL script (multiple SQL commands)
`close`	Closes the cursor object

Table 4.3: Cursor object methods

Python Type	SQLite Type
`None`	`NULL`
`int`	`INTEGER`
`long`	`INTEGER`
`float`	`REAL`
`str`	`TEXT`
`buffer`	`BLOB`

Table 4.4: Python and SQLite types mapping

The cursor object has several useful methods (Table 4.3). Through the cursor, we will execute all of our SQL commands.

Before creating a table, we need to understand how SQLite stores information in a database. SQLite uses five native data types (a simplified system from other SQL database managers). Table 4.4, gives a mapping between Python and SQLite native types.

Creating and Dropping Tables

Let's create a table.

```
cur.execute('CREATE TABLE StudentInformation (StudentID INTEGER NOT NULL, Name ↩
    TEXT, MajorCode INTEGER);')
```

This will create the empty table in Table 4.5.

The arguments in parentheses are the column names followed by the data type that entries in that column will be, and together these form the schema of the table. The `INTEGER` data type in SQLite is a 1, 2, 3, 4, 6, or 8 byte integer depending on the value. The `NOT NULL` command is a *constraint* on the StudentID column. It requires that all records in the table have a student ID.

> **Note**
>
> SQLite does not enforce types on columns. Just like Python, SQLite is dynamically typed. However, most other database systems strictly enforce column types. It is a good idea to conform to the column types specified in the schema.

StudentID	Name	MajorCode

Table 4.5: StudentInformation

Note that each command we execute returns the same cursor object. This object is equipped with a method that allows us to look at any results of the previous command. The result is formally known as the *result set*. If you use `cur.fetchall()`, you will see an empty list. That is because the create table command does not return a result set.

Now we want to build a relation between students, the courses they've had, and their grades in those courses.

```
cur.execute('CREATE TABLE StudentGrades (StudentID INT NOT NULL, CourseID INT, ↩
    Grade TEXT);')
```

Problem 1. In this problem, you will create two new tables in the database "sql1". The first table will be called MajorInfo and have a column called MajorID and MajorName. MajorID is an integer and MajorName is a string.

The second table will be called CourseInfo and have columns called CourseID and CourseName, also integers and strings, respectively.

Hint: In order to view information about the columns of the table, run the following command:

```
cur.execute("PRAGMA table_info('table_name')")
for info in cur:
    print info
```

For each column, this command will output (ID, name, type, notnull, default value, primary key). Also, don't forget to commit and close your database.

We can also destroy tables using the `DROP TABLE` command.

```
cur.execute("CREATE TABLE test_table (id INT, name TEXT);")
```

We can delete the table by dropping it.

```
cur.execute("DROP TABLE test_table;")
```

If a table doesn't exist, an exception will be raised. We can tell the database to drop the table only if it really exists by using `DROP TABLE IF EXISTS test_table;`.

Inserting and Removing Data

Let's insert some data into our new tables. We can add rows to tables using the `INSERT INTO` command.

```
cur.execute("INSERT INTO StudentInformation VALUES(55, 'John Smith', 2);")
```

After running this statement, we will have the table in Table 4.6.

Note that SQLite will assume that values match sequentially with the schema of the table. We can also specify the schema of the table to use in the mapping of the values.

StudentID	Name	MajorCode
55	John Smith	2

Table 4.6: StudentInformation

```
cur.execute("INSERT INTO StudentInformation(MajorCode, Name, StudentID)  VALUES↩
    (55, 'John Smith', 2);")
```

This will map the value 55 to MajorCode and the value 2 to StudentID. This may be useful sometimes.

It can quickly become tedious to insert large amounts of data into a table, one row at a time. We can automate the process somewhat by using the `executemany` method of the cursor object. To insert several rows into a table using a single command, we can do the following:

```
cur.executemany("INSERT INTO StudentInformation VALUES (?, ?, ?, ?);", rows)
```

In the code above, we assume that `rows` is a Python list of tuples, each tuple containing the data for one row.

We may remove rows from a table using the `DELETE FROM` command.

```
cur.execute("DELETE FROM StudentInformation WHERE MajorCode=55;")
```

Achtung!

Never use Python's string operations to construct a SQL query. It is extremely insecure and is an easy target for a well known type of database called a SQL injection attack.

Parameter substitution can be used to construct dynamic queries. In the simplest way, it involves using a '?' character whenever you want to use a value and providing a sequence of values as a second argument to `execute()`.

```
statement = "INSERT INTO StudentInformation VALUES(?, ?, ?, ?);"
values = (55, 'John Smith', 372897382, 2)
cur.execute(statement, values)
```

Problem 2. The ICD is a large collection of codes used to classify any diagnosis that a doctor would make. When someone goes to the hospital or doctors office, their visit will be recorded using these codes. Insurance companies, the government, and researchers find this data useful. The data file provided to you, `icd9.csv`, has simulated health histories for one million persons. Each line has columns for identification number, gender, and age, followed by ICD-9 codes of various quantities. Note that the codes for each individual are written in a single string, each code separated by semicolons. Create a new database with a single table to store all the simulated data. Call the database "sql2" and the table "ICD." Your table should have four columns, one each for id number, gender, age, and codes.

Because of the volume of data, it is highly recommended you use the `executemany()` method of the cursor. It will be about twice as fast as using an `execute()` for each line of the CSV file. Recall the `csv` package in Python. To read a CSV file into a list of tuples, where each tuple consists of the delimited values of a particular line in the file, one can use the following code as a guideline:

```
import csv
with open('filename', 'rb') as csvfile:
    rows = [row for row in csv.reader(csvfile, delimiter=',')]
```

Hint: Don't forget to commit and close your database.

Problem 3. Create the following tables in the same database you created in Problem 1 ("sql1"). You may do so however you think is best.

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

Table 4.7: StudentInformation

ID	Name
1	Math
2	Science
3	Writing
4	Art

Table 4.8: MajorInfo

StudentID	ClassID	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-

Table 4.9: StudentGrades

ClassID	Name
1	Calculus
2	English
3	Pottery
4	History

Table 4.10: CourseInfo

Updating Rows of a Table

We can modify records in a table by using the `UPDATE` command.

```
cur.execute("UPDATE StudentInformation SET MajorCode=2, StudentID=55, Name='↩
    Jonathan Smith' WHERE StudentID=2;")
```

NOTE

When updating a table, having a sufficient `WHERE` clause is essential. Any record that matches the criteria will be modified. If we omitted the `WHERE` clause, every record in the table would be set to the values given in the example.

Adding Columns to a Table

After you have created a table, you can add more columns to the table by using the `ALTER TABLE` command. For example, if we wanted to add the column "age" into our students table. We run this command.

```
cur.execute("ALTER TABLE StudentInformation ADD COLUMN age INTEGER;")
```

Selecting Data From Tables

The process of retrieving data from a table in a database is accomplished by the `SELECT` statement. The `SELECT` statement can be thought of as a very high level set description. For example, to view the contents of an entire table, we simply need to unconditionally select its contents.

```
SELECT * FROM students;
```

This is equivalent to the following set (where x is a row).

$$\{x : x \in \text{classes}\}$$

We can also select specific columns.

```
SELECT StudentID, Name FROM students;
```

Or we can impose conditions on the selected rows.

```
SELECT StudentID, Name FROM students WHERE MajorCode=1;
```

This query results in the following table (Table 4.11) where the contents are all the students that are math majors.

You can also further refine which rows you select by using the `AND` or `OR` commands. These commands will connect expressions. For example, to get the math and science majors. One could run the command.

```
SELECT StudentID, Name FROM students WHERE MajorCode=1 OR MajorCode=2;
```

Select statements return a *result set*. This is an iterable object. Each row in the object is represented as a tuple of values.

```
cur.execute('SELECT StudentID, Name FROM students WHERE MajorCode=1;')
for student in cur:
    print student
```

StudentID	Name
401767594	Michelle Fernandez
678665086	Gilbert Chapman
341324754	Cassandra Holland

Table 4.11: Selected students who are math majors.

Method	Description
`fetchone()`	Return a single row from the result set
`fetchmany(n)`	Return the next n rows from the result set
`fetchall()`	Return the entire result set

Table 4.12: Fetch methods of a cursor.

We can also use the fetch methods of the returned cursor to extract rows from the result set (Table 4.12).

Problem 4. From the ICD9 table you created in Problem 2, how many men between the ages of 25 and 35 are there? How many women between those same ages? Return your answers as a tuple.

When an Error Occurs

It is important to be able to recover from errors gracefully, especially when working a database. Data integrity in a database is often a critical need. When an error occurs, we need to undo the changes that triggered the error. Fortunately, `sqlite3` reports a variety of errors. These errors and when they are raised is explained in PEP249 (`http://legacy.python.org/dev/peps/pep-0249/`).

Error The base class for errors thrown by `sqlite3`. All other errors inherit from this class. Catching this error will catch any error raised.

InterfaceError Raised when there is a problem with the interface to the database rather than the database itself.

DatabaseError Raised when there is an error with the database itself.

DataError Subclass of DatabaseError. Raised when there are errors in the processed data (division by zero, value out of range, etc.).

OperationalError Subclass of DatabaseError. Raised for errors related to the database that are not the fault of the programmer. For example, an unexpected disconnect, failure to process a transaction, a memory allocation error during a transaction, etc.

IntegrityError Subclass of DatabaseError. Raised when the relational integrity of the database is compromised.

InternalError Subclass of DatabaseError. Raised when there is an internal error such as an invalid cursor, out-of-sync transaction, etc.

ProgrammingError Subclass of DatabaseError. Raised for programming errors.

NotSupportedError Subclass of DatabaseError. Raised when a method is called that is not supported by the database.

The way to gracefully recover from errors is to catch them and handle them accordingly. For example, if any error occurs, with the interface or the database, we immediately rollback the transaction. If no error occurs, commit. We could use if-statements or we could use a try-except block.

```
try:
    <code>
    db.commit()
except sql.Error:
    db.rollback()
```

Note that rolling back is not needed if we are just performing queries. If we don't change any of the data in the database, there is no need to roll anything back. However, even with queries, there is the potential for errors. You must design your code to handle these errors gracefully.

5 SQL 2 (The SQL Sequel)

Lab Objective: *Learn more of the advanced and specialized features of SQL.*

Database Normalization

Normalizing a database is the process of organizing tables and columns to minimize the amount of redundant information in the database. For example, a non-normalized database might have a table that stores customer contact information and a table that contains all of the products a company has sold. However, they might want to track who buys what products in case they need to contact them later. To do so, they store all the contact information of a particular buyer along with every item they purchased. Now, two tables store the customer contact information. If we needed to update a customer's phone number, we have to update two tables. While that may not be bad for small databases, larger databases would be near impossible to update correctly. The idea of normalizing a database allows us to store all customer contact information in one place in the database. All other tables that might need a customer's name, phone number, or address would reference the contact information table. When an update needs to be performed, we only need to update the contact information table. Then any table that references this information is also automatically up to date.

To properly normalize a database, we need to discuss the types of relations tables might have.

One to One

This is the simplest relation to model. A single table can be used to express this relation. The relation is between one record and at most one other record. An example of this relationship is an employee and their organization. One employee works at one organization. Another example would be that a driver's license belongs to only one person.

One to Many

This relationship and its inverse must be modeled with at least two tables. The general approach is to use a unique ID. Note that a relationship that appears one to one may actually be a one to many relationship. Many people will, therefore, use the same unique ID approach on one to one relationships too in the case it turns out to be a one to many relationship. An example of a one to many relationship would be between an department and its employees. The department would receive a unique ID and then each employee in that department would be tagged with that ID.

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

Table 5.1: students

ID	Name
1	Math
2	Science
3	Writing
4	Art

Table 5.2: fields

Many to Many

This relationship requires at least three tables. A many to many relationship can be visualized as two, separate one to many relationships. The records in each of the two tables receive a unique ID. A third table then serves as a map between IDs of table to IDs of the other table. An example of a many to many relationship is doctors and patients. One doctor can have several patients and one patient can have several doctors.

For the rest of the lab, we will be using the following tables: 5.1, 5.2, 5.3, and 5.4.

Problem 1. Classify the relations between the various records in these tables: 5.1, 5.2, 5.3, and 5.4.

Classify each relation as either one to one, one to many, or many to many. Identify the tables used in each relationship.

Note

There are instances where you would not want a completely normalized database. Whether to normalize your database depends on your specific needs. Usually, though, the decision to denormalize a database is a last-resort attempt to improve performance.

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

Table 5.3: grades

ClassID	Name
1	Calculus
2	English
3	Pottery
4	History

Table 5.4: classes

Joining Tables

We can use SQL to join two or more tables together for a query. This is a very powerful tool. SQLite supports three types of standard table joins.

Joining tables is a common practice to collect data from different parts of the database into a single table. Joins are absolutely essential in a normalized database since data is split between multiple tables.

Inner Join

This is often the default join operation in SQL. An inner join can be depicted as an intersection of two or more tables. When performing an inner join on tables, the result will only be those records that match across all tables. For example, this join will select all the students' names along with their major as long as the students.majorcode matches the majors.id. If the students.majorcode is missing or null, then they won't be selected.

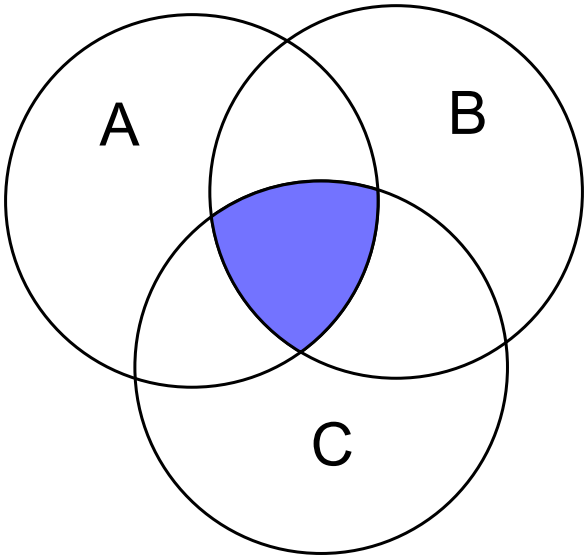


Figure 5.1: An inner joining of tables A, B, and C.

students.name	majors.name
Michelle Fernandez	Math
Roberta Cook	Science
Rene Cross	Writing
Cameron Kim	Art
Kristopher Tran	Science
Cassandra Holland	Math
Sammy Burke	Science

Table 5.5: An inner join of students and majors

```
SELECT students.name, majors.name FROM students JOIN majors ON students.↩
    majorcode=majors.id;
```

An inner join is equivalent to the following pseudo-loop in Python

```
for row_s in students:
    for row_m in majors:
        if predicates(row_s, row_m):
            yield columns(row_s, row_m)
```

Left Outer Join

A left outer join will return all relations from the left table even if they don't match any relation on the joined tables. An illustration of a left outer join is given in figure 5.2.

A Pythonesque loop that illustrates how to perform a left outer join is

```
for row_s in students:
```

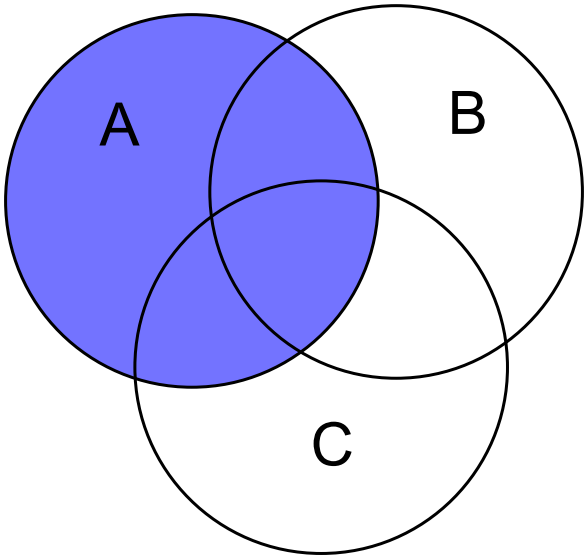


Figure 5.2: A left outer table join of A with tables B and C.

students.name	majors.name
Michelle Fernandez	Math
Gilbert Chapman	None
Roberta Cook	Science
Rene Cross	Writing
Cameron Kim	Art
Mercedes Hall	None
Kristopher Tran	Science
Cassandra Holland	Math
Alfonso Phelps	None
Sammy Burke	Science

Table 5.6: A left outer join of students and majors

```
    for row_m in majors:
        if predicates(row_s, row_m):
            yield columns(row_s, row_m)
        else:
            yield columns(row_s)
```

The following left outer join will result in the table shown in table 5.6.

```
SELECT students.name, majors.name FROM students LEFT OUTER JOIN majors ON ↩
    students.majorcode=majors.id;
```

Function	Description
MIN()	Retrieve the smallest numeric value of a column
MAX()	Retrieve the largest numeric value of a column
SUM()	Sum the numeric values of a column
AVG()	Retrieve the average numeric value of the column
COUNT()	Retrieve the total number of matching records in a column

Table 5.7: SQL aggregation functions

Cross Join

Essentially a Cartesian product of tables. Care must be taken when using cross join because of the size of the joined table. A cross join should only be used on small tables. It matches each relation in one table with every other possible combination of relations in the joined tables. For example, if you were to run a cross join on the above join. You would get every student matched with every major. This doesn't really make much sense but can be useful for other applications. Use with caution.

Advanced Selections

Aggregate functions are useful for summarizing the data in a column. The functions can be found in 5.7.

We can count the number of students by executing the following SQL statement.

```
SELECT COUNT(*) FROM students;
```

Ordering and Grouping Relations

The `ORDER BY` keyword can be used to sort the result set by columns. We can sort in ascending order or descending order.

```
SELECT name FROM students ORDER BY name ASC;
SELECT name FROM students ORDER BY name DESC;
```

Another useful SQL keyword is the `GROUP BY` keyword. It is used along with an aggregating function to group the result set by columns.

```
SELECT grade, COUNT(studentid) FROM grades GROUP BY grade;
```

The result set is given in table 5.8.

Problem 2. Create a database containing tables 5.1, 5.2, 5.3, and 5.4. Write a SQL query to count how many students belong to each major, including students that don't have a major. Sort your results in ascending order by name. Your result set should be table 5.9

grade	COUNT(studentid)
None	5
A	4
A+	1
A-	1
B	2
B-	1
C	3
C+	1
C-	1
D	1
D-	1

Table 5.8: Grouping of students by grade.

None	3
Art	1
Math	2
Science	3
Writing	1

Table 5.9: Result set

Another important keyword is the `HAVING` keyword. This is necessary because the `WHERE` clause does not support aggregate functions. A `HAVING` clause requires a `GROUP BY` clause. The following will not work.

```
SELECT grade FROM grades GROUP BY grade WHERE COUNT(*)=1;
```

Since `COUNT` is an aggregating function, the following is required.

```
SELECT grade FROM grades GROUP BY grade HAVING COUNT(*)=1;
```

This SQL query returns all the grades that occur only once in the table. A simple way to remember the difference is *`WHERE` operates on individual records and `HAVING` operates on groups of records.*

Problem 3. Select all the students who have received grades (non-null grades) in more than two classes. How many grades did he receive?

Case Expression

A case expression allows you to temporarily modify records from a select operation. There are two forms of the case expression; simple and searched. The simple form of the expression is a match and replace on a specified column. A simple case expression is demonstrated below. This code will return the name of the student along with their majorcode. But instead of integers, the names of the majors will be returned.

```
SELECT name,
CASE majorcode
    WHEN 1 THEN 'Math'
    WHEN 2 THEN 'Science'
    WHEN 3 THEN 'Writing'
    WHEN 4 THEN 'Art'
    ELSE 'Undeclared'
    END AS major
FROM students;
```

A searched case expression using a boolean expression for the `WHEN` clauses.

```
SELECT name,
CASE
    WHEN majorcode IS NULL THEN 'Undeclared'
    ELSE majorcode
    END AS major,
CASE
    WHEN minorcode IS NULL THEN 'Undeclared'
    ELSE minorcode
    END AS minor
FROM students;
```

Problem 4. Find the overall GPA of all the students in the school. Use a regular 4.0 scale (A=4.0, B=3.0, C=2.0, D=1.0). Any pluses or minuses are dropped so an A- becomes an A.

Your result set should be one column and one row with average of all GPAs of all the students taking classes. Your solution should return a single floating point number.

Use the `ROUND()` function in SQL to round your result to the nearest hundredth.

Like Command

The SQL keyword, `LIKE`, allows us to match patterns in a column. For example,

```
SELECT name, studentid FROM students WHERE studentid LIKE '\%4';
```

will return all the students that have a student ID that ends with the digit 4. Use '\%' before the '4' to signify that there can be any number of characters before the '4.' If you only want one character, you can use an underscore. For example, if you were to search a database of words in the english dictionary and you entered the following command:

```
SELECT word FROM englishDictionary WHERE word LIKE 'i_';
```

You would get words like 'is,' 'it,' or 'in.'

Problem 5. Write a SQL statement that will find all students with a last name that begins with the letter 'C' and return their names and majors. Your returned records should be

Gilbert Chapman	None
Roberta Cook	Science
Rene Cross	Writing

6 Pandas I: Introduction to Pandas

Lab Objective: *Become acquainted with the data structures and tools that pandas offers for data analysis.*

In volumes 1 and 2, we solved data problems primarily using NumPy and SciPy. While extremely flexible and useful tools, these libraries lack some of the high-level data-analytic abstractions present in other popular data packages. In particular, *pandas* is a Python library that is more specifically built for data analysis. In this lab, we will give an overview of some of the functions which will prove very useful in this regard.

Data Structures in pandas

Just as NumPy is built on the `ndarray` data structure suited for efficient scientific and numerical computation, pandas is centered around a handful of core data structures custom built for data analysis. These data structures include the `Series`, `DataFrame`, and `Panel`, which correspond roughly to one, two, and three-dimensional arrays. We will explore the first two data structures in some detail. The interested reader can learn more about the `Panel` data structure (the least-used one in pandas) in the online documentation; we will not delve into this data structure here.

Series

The `Series` is a one-dimensional array with labeled entries. The values contained may be any data type, including integers, strings, or general Python objects. Further, unlike NumPy arrays, pandas `Series` need not be homogeneous. That is, they can hold values of different data types. Together, the values are referred to as the *data* of the `Series`. The labels must consist of hashable types, and are most commonly integers or strings. Together, the labels are referred to as the *index* of the `Series`. Thus, a `Series` consists of data and an index. The most basic way to initialize such an object is as follows:

```
>>> import pandas as pd
>>> s = pd.Series(data, index=index)
```

We don't need to explicitly define an index. The default is `np.arange(len(data))`. For example, we can create a `Series` containing the integers from 9 to 0:

```
>>> s1 = pd.Series(np.arange(9, -1, -1))
>>> s1.values    #the data
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
>>> s1.index     #the labels
Int64Index([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype='int64')
>>> s1           #left column is index, right column is data
0    9
1    8
2    7
3    6
4    5
5    4
6    3
7    2
8    1
9    0
dtype: int64
```

Here is an example where we create customized labels:

```
>>> import numpy as np
>>> data = np.random.randn(3)
>>> index = ['first', 'second', 'third']
>>> s2 = pd.Series(data, index=index)
>>> s2
first     1.661255
second   -0.033570
third    -2.185991
dtype: float64
```

We can create a `Series` having constant values in the following manner:

```
>>> val = 4     #desired constant value of Series
>>> n = 6       #desired length of Series
>>> s3 = pd.Series(val, index=np.arange(n))
>>> s3
0    4
1    4
2    4
3    4
4    4
5    4
dtype: int64
```

It is also possible to use a Python dictionary to create a `Series`:

```
>>> d = {'e1': 93, 'e2': 95, 'e3': 87, 'e4': 82, 'e5': 94}
>>> s4 = pd.Series(d)
```

```
>>> s4
e1    93
e2    95
e3    87
e4    82
e5    94
dtype: int64
```

Note that we didn't need to specify the index; the keys of the dictionary are used as the index for the `Series`. There are many more ways to create `Series` objects. For a more complete discussion of how to create `Series` objects, see the online documentation.

Problem 1. Create the following pandas `Series`.

- Constant array with value -3, length 5. The index labels should be the first five positive even integers.
- Data is given by the dictionary:
  `{'Bill': 31, 'Sarah': 28, 'Jane': 34, 'Joe': 26}`.

DataFrame

The `DataFrame` data structure is a two-dimensional generalization of the `Series`. It can be viewed as a tabular structure with labeled rows and columns. The row labels are collectively called the index, and the column labels are collectively called the columns. An individual column in a `DataFrame` object is a `Series`.

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

```
>>> x = pd.Series(np.random.randn(4), ['a', 'b', 'c', 'd'])
>>> y = pd.Series(np.random.randn(5), ['a', 'b', 'd', 'e', 'f'])
>>> d = {1: x, 2: y}
>>> df1 = pd.DataFrame(d)
>>> df1
          1         2
a -0.924259 -0.708301
b  0.767422 -2.214516
c  0.399212       NaN
d  0.130365 -2.352364
e       NaN  0.789419
f       NaN -0.859482
```

Note that the index of this `DataFrame` is the union of the index of `Series x` and that of `Series y`. The columns are given by the keys of the dictionary `d`. Since `x` doesn't have a label `e`, the value in row `e`, column `1` is `NaN`. This same reasoning explains the other missing values as well. Note that if we take the first column of the `DataFrame` and drop the missing values, we recover the `Series x`:

```
>>> x == df1[1].dropna()
a    True
b    True
c    True
d    True
dtype: bool
```

Achtung!

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

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

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

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

As with Series, if we don't specify the index or columns, the default is `np.range(n)`, where `n` is either the number of rows or columns.

It is also possible to create multi-indexed arrays, for example:

```
>>> grade=['eighth', 'ninth', 'tenth']
>>> subject=['math', 'science', 'english']
>>> myindex = pd.MultiIndex.from_product([grade, subject], names=['grade', '↩
    subject'])
>>> myseries = pd.Series(np.random.randn(9), index=myindex)
>>> myseries
grade   subject
eighth  math       1.706644
        science   -0.899587
        english   -1.009832
ninth   math       2.096838
        science    1.884932
        english    0.413266
tenth   math      -0.924962
        science   -0.851689
        english    1.053329
dtype: float64
```

Multi-indexing is visually convenient, and will be explored further in Lab 8, where we will discuss pandas pivot tables.

Data I/O

Being able to import and export data is a fundamental skill in data science. Unfortunately, with the multitude of data formats and conventions out there, importing data can often be a tricky task. The pandas library seeks to reduce some of the difficulty by providing file readers for various types of formats, including CSV, Excel, HDF5, SQL, JSON, HTML, and pickle files.

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

- `delimiter`: This argument specifies the character that separates data fields, often a comma or a whitespace character.
- `header`: The row number (starting at 0) in the CSV file that contains the column names.
- `index_col`: If you want to use one of the columns in the CSV file as the index for the `DataFrame`, set this argument to the desired column number.
- `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 assigned to this argument.

There are several other keyword arguments, but this should be enough to get you started.

When you need to save your data, pandas allows you to write to several different file formats. A typical example is the `to_csv()` function method attached to `Series` and `DataFrame` objects, which writes the data to a CSV file. Keyword arguments allow you to specify the separator character, omit writing the columns names or index, and specify many other options. The code below demonstrates its typical usage:

```
>>> df.to_csv("my_df.csv")
```

Viewing and Accessing Data

Once we have our data ready to go in pandas, how can we interact with it? In this section we will explore some elementary accessing and querying techniques that enable us to maneuver through and gain insight into our data.

Basic Data Access

Some of the basic slicing paradigms in NumPy carry over to pandas. For example, we can slice a `Series` using the usual syntax:

```
>>> s = pd.Series(np.random.randn(5))
>>> s[1:3]

1    3.188112
2    0.080191
dtype: float64
```

Notice that both the data and the index are sliced in this manner.

Likewise, we can slice the rows of a `DataFrame` much as with a NumPy array:

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

          I        II
a  0.758867  1.231330
b  0.402484 -0.955039

[2 rows x 2 columns]
```

More generally, we can select subsets of the data using the `.iloc` and `.loc` indexers. The `.loc` index selects rows and columns based on their *labels*, while the `.iloc` method selects them based on their integer *position*. Accessing `Series` and `DataFrame` objects using these indexing operations is more efficient than using bracket indexing, because the bracket indexing has to check many cases before it can determine how to slice the data structure. By using `loc`/`iloc` explicitly, you bypass the extra checks.

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

a    1.231330
c    0.556121
Name: II, dtype: float64

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

c   -0.475952
d   -0.518989
Name: I, dtype: float64
```

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

```
>>> # get second column of df
>>> df['II']

a    1.231330
```

```
b   -0.955039
c    0.556121
d    0.173165
Name: II, dtype: float64

>>> # equivalent result to above
>>> df.II

a    1.231330
b   -0.955039
c    0.556121
d    0.173165
Name: II, dtype: float64
```

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

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

a    0
b    0
c    0
d    0
Name: II, dtype: int64

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

           I  II  III
a   -0.460457   0    1
b    0.973422   0    1
c   -0.475952   0    1
d   -0.518989   0    1
```

Plotting

Plotting is often a much more effective way to view and gain understanding of a dataset than simply viewing the raw numbers. Fortunately, pandas interfaces well with matplotlib, allowing relatively painless data visualization.

Most of the functionality of plotting using pandas will be discussed in Lab 7. In this lab, we will simply show how to plot a `Series`. Doing so is easy, as the `Series` object is equipped with its own plot function.

Let's start with visualizing a simple random walk. By way of background, a *random walk* is a stochastic process used to model a non-deterministic path through some space. It is a useful construct in many fields and can be used to explain things like the motion of a molecule as it travels through a liquid to modeling the fluctuations of stock prices. Here we will simulate a one-dimensional symmetric random walk on the integers, which can be described as follows.

1. Start at 0.
2. Flip a fair coin.
3. If heads, move one unit to the right. Otherwise, move one unit to the left.
4. Go to Step 2.

How can we simulate this random walk efficiently? Note that the walk is really characterized by the outcomes of the coin flip. If we represent heads by the number 1 and tails by -1, then our position at a given moment is just the cumulative sum of all previous outcomes. Below, we simulate a sequence of coin flips, build the resulting random walk, and plot the outcome.

```
>>> import matplotlib.pyplot as plt
>>> N = 1000        # length of random walk
>>> s = np.zeros(N)
>>> s[1:] = np.random.binomial(1, .5, size=(N-1,))*2-1 #coin flips
>>> s = pd.Series(s)
>>> s = s.cumsum()  # random walk
>>> s.plot()
>>> plt.ylim([-50, 50])
>>> plt.show()
>>> plt.close()
```

The random walk is shown in Figure 6.1.

Problem 2. Create five random walks of length 100, and plot them together in the same plot.

Next, create a "biased" random walk by changing the coin flip probability of head from 0.5 to 0.51. Plot this biased walk with lengths 100, 10000, and then 100000. Notice the definite trend that emerges. Your results should be comparable to those in Figure 6.2.

SQL Operations in pandas

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

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

```
>>> #build toy data for SQL operations
>>> name = ['Mylan', 'Regan', 'Justin', 'Jess', 'Jason', 'Remi', 'Matt', '↩
    Alexander', 'JeanMarie']
```

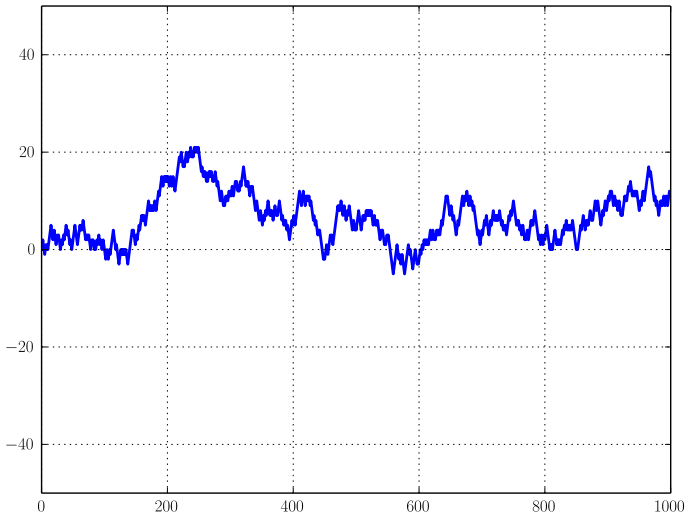


Figure 6.1: Random walk of length 1000.

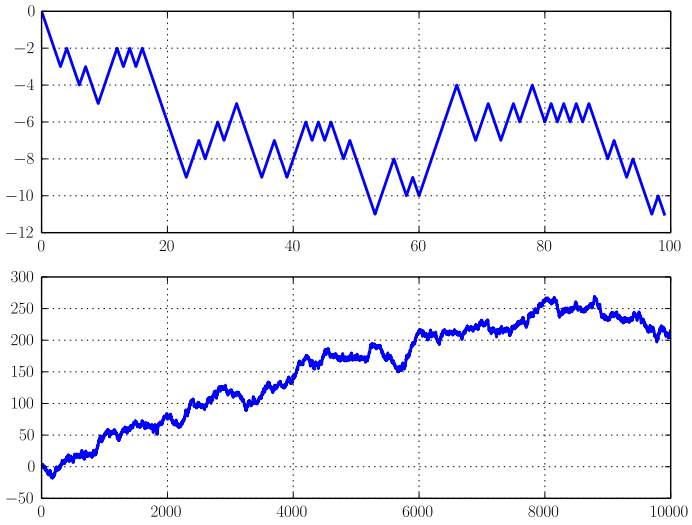


Figure 6.2: Biased random walk of length 100 (above) and 10000 (below).

```
>>> sex = ['M', 'F', 'M', 'F', 'M', 'F', 'M', 'M', 'F']
>>> age = [20, 21, 18, 22, 19, 20, 20, 19, 20]
>>> rank = ['Sp', 'Se', 'Fr', 'Se', 'Sp', 'J', 'J', 'J', 'Se']
>>> ID = range(9)
>>> aid = ['y', 'n', 'n', 'y', 'n', 'n', 'n', 'y', 'n']
>>> GPA = [3.8, 3.5, 3.0, 3.9, 2.8, 2.9, 3.8, 3.4, 3.7]
```

```
>>> mathID = [0, 1, 5, 6, 3]
>>> mathGd = [4.0, 3.0, 3.5, 3.0, 4.0]
>>> major = ['y', 'n', 'y', 'n', 'n']
>>> studentInfo = pd.DataFrame({'ID': ID, 'Name': name, 'Sex': sex, 'Age': age,↩
    'Class': rank})
>>> otherInfo = pd.DataFrame({'ID': ID, 'GPA': GPA, 'Financial_Aid': aid})
>>> mathInfo = pd.DataFrame({'ID': mathID, 'Grade': mathGd, 'Math_Major': major↩
    })
```

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

```
>>> mathInfo.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 5 entries, 0 to 4
Data columns (total 3 columns):
Grade         5 non-null float64
ID            5 non-null int64
Math_Major    5 non-null object
dtypes: float64(1), int64(1), object(1)
```

We can also get some basic information about the structure of the `DataFrame` using the `head()` or `tail()` methods.

```
>>> mathInfo.head()
   Grade  ID Math_Major  ID  Age  GPA
0    4.0   0          y   0   20  3.8
1    3.0   1          n   2   18  3.0
2    3.5   5          y   4   19  2.8
3    3.0   6          n   6   20  3.8
4    4.0   3          n   7   19  3.4
```

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

```
>>> # SELECT ID, Age FROM studentInfo
>>> studentInfo[['ID', 'Age']]

>>> # SELECT ID, GPA FROM otherInfo WHERE Financial_Aid = 'y'
>>> otherInfo[otherInfo['Financial_Aid']=='y'][['ID', 'GPA']]

>>> # SELECT Name FROM studentInfo WHERE Class = 'J' OR Class = 'Sp'
>>>  studentInfo[studentInfo['Class'].isin(['J','Sp'])]['Name']
```

Problem 3. The example above shows how to implement a simple WHERE condition, and it is easy to have a more complex expression. Simply enclose each condition by parentheses, and use the standard boolean operators `&` (AND), `|` (OR), and `~` (NOT) to connect the conditions appropriately. Use pandas to execute the following query:

```
SELECT ID, Name from studentInfo WHERE Age > 19 AND Sex = 'M'
```

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

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

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

Problem 4. Using a join operation, create a `DataFrame` containing the ID, age, and GPA of all male individuals. You ought to be able to accomplish this in one line of code.

Be aware that other types of SQL-like operations are also possible, such as UNION. When you find yourself unsure of how to carry out a more involved SQL-like operation, the online pandas documentation will be of great service.

Analyzing Data

Although pandas does not provide built-in support for heavy-duty statistical analysis of data, there are nevertheless many features and functions that facilitate basic data manipulation and computation, even when the data is in a somewhat messy state. We will now explore some of these features.

Basic Data Manipulation

Because the primary pandas data structures are subclasses of the `ndarray`, they are valid input to most NumPy functions, and can often be treated simply as NumPy arrays. For example, basic vectorized operations work just fine:

```
>>> x = pd.Series(np.random.randn(4), index=['a', 'b', 'c', 'd'])
>>> y = pd.Series(np.random.randn(5), index=['a', 'b', 'd', 'e', 'f'])
>>> x**2
a    1.710289
b    0.157482
c    0.540136
d    0.202580
dtype: float64
>>> z = x + y
>>> z
a    0.123877
b    0.278435
c         NaN
d   -1.318713
e         NaN
f         NaN
dtype: float64
>>> np.log(z)
a   -2.088469
b   -1.278570
c         NaN
d         NaN
e         NaN
f         NaN
dtype: float64
```

Notice that pandas automatically aligns the indexes when adding two `Series` (or `DataFrames`), so that the the index of the output is simply the union of the indexes of the two inputs. The default missing value `NaN` is given for labels that are not shared by both inputs.

It may also be useful to transpose `DataFrames`, re-order the columns or rows, or sort according to a given column. Here we demonstrate these capabilities:

```
>>> df = pd.DataFrame(np.random.randn(4,2), index=['a', 'b', 'c', 'd'], columns↩
    =['I', 'II'])
>>> df
          I        II
a -0.154878 -1.097156
```

```
b -0.948226  0.585780
c  0.433197 -0.493048
d -0.168612  0.999194

[4 rows x 2 columns]

>>> df.transpose()
          a         b         c         d
I  -0.154878 -0.948226  0.433197 -0.168612
II -1.097156  0.585780 -0.493048  0.999194

[2 rows x 4 columns]

>>> # switch order of columns, keep only rows 'a' and 'c'
>>> df.reindex(index=['a', 'c'], columns=['II', 'I'])
         II         I
a -1.097156 -0.154878
c -0.493048  0.433197

[2 rows x 2 columns]

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

[4 rows x 2 columns]
```

Basic Statistical Functions

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

```
>>> df.describe()
              I        II
count  4.000000  4.000000
mean  -0.209630 -0.001308
std    0.566696  0.964083
min   -0.948226 -1.097156
25%   -0.363516 -0.644075
50%   -0.161745  0.046366
75%   -0.007859  0.689133
max    0.433197  0.999194
```

```
[8 rows x 2 columns]
```

Functions for calculating means and variances, the covariance and correlation matrices, and other basic statistics are also available. Below, we calculate the means of each row, as well as the covariance matrix:

```
>>> df.mean(axis=1)
a   -0.626017
b   -0.181223
c   -0.029925
d    0.415291
dtype: float64

>>> df.cov()
           I        II
I   0.321144 -0.256229
II -0.256229  0.929456

[2 rows x 2 columns]
```

Dealing with Missing Data

Missing data is a ubiquitous problem in data science. Fortunately, pandas is particularly well-suited to handling missing and anomalous data. As we have already seen, the pandas default for a missing value is `NaN`. In basic arithmetic operations, if one of the operands is `NaN`, then the output is also `NaN`. The following example illustrates this concept:

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

If we are not interested in the missing values, we can simply drop them from the data altogether:

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

This is not always the desired behavior, however. It may well be the case that missing data actually corresponds to some default value, such as zero. In this case, we can replace all instances of `NaN` with a specified value:

```
>>> # fill missing data with 0, add
>>> x.fillna(0) + y
0    0.731521
1    0.623651
2    2.396344
3    1.829400
4    3.351182
dtype: float64
```

Other functions, such as `sum()` and `mean()` treat NaN as zero by default. When dealing with missing data, make sure you are aware of the behavior of the pandas functions you are using.

Problem 5. Using the dataset contained in the file `crime_data.txt` and the techniques learned in this lab, use pandas to complete the following.

- Load the data into a pandas `DataFrame`, using the column names in the file and the column titled "Year" as the index. Make sure to skip lines that don't contain data.
- Insert a new column into the data frame that contains the crime rate by year (the ratio of "Total" column to the "Population" column).
- Plot the crime rate as a function of the year.
- List the 5 years with the highest crime rate in descending order.
- Calculate the average number of total crimes as well as burglary crimes between 1960 and 2012.
- Find the years for which the total number of crimes was below average, but the number of burglaries was above average.
- Plot the number of murders as a function of the population.
- Select the Population, Violent, and Robbery columns for all years in the 1980s, and save this smaller data frame to a CSV file `crime_subset.txt`.

7 Pandas II: Plotting with Pandas

Lab Objective: *Pandas has many built-in plotting methods that wrap around matplotlib. Since pandas provides tools for organizing and correlating large sets of data, it is important to be able to visualize these relationships. First, we will go over different types of plots offered by pandas, and then some techniques we can use to visualize data in useful ways.*

Plotting Data Frames

Recall from Lab 6 that in Pandas, a *DataFrame* is an ordered collection of *Series*. A Series is similar to a dictionary, with values assigned to various labels, or indices. Each Series becomes a column in the data frame, with each row corresponding to an index. When several Series are combined into a single data frame, it becomes very easy to compare and visualize data. Each entry has an associated index and column.

DataFrames have several methods for easy plotting. Most are simple wrappers around matplotlib commands, but some produce styles of plots that we have not previously seen. In this lab, we will go over seven different types of plots offered by pandas.

For these examples, we will use the data found in the file `crime_data.txt`.

```
>>> import pandas as pd
>>> crime = pd.read_csv("crime_data.txt", header=1, index_col=0)
```

Line Plots

In Lab 6, we showed how to plot a `Series` against its index (the years, in this case) using matplotlib. With a few extra lines we modify the x-axis label and limits and create a legend.

```
>>> from matplotlib import pyplot as plt
>>> plt.plot(crime["Population"], label="Population")
>>> plt.xlabel("Year")
>>> plt.xlim(min(crime.index), max(crime.index))
>>> plt.legend(loc="best")
>>> plt.show()
```

Equivalently, we can produce the exact same plot with a single line using the `DataFrame` method `plot()`. Specify the y-values as a keyword argument. The x-values default to the index of the Series. See Figure 7.1.

```
>>> crime.plot(y="Population")
>>> plt.show()
```

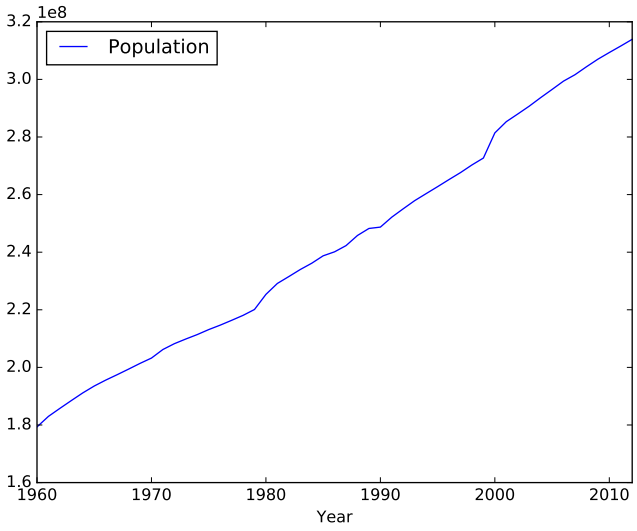


Figure 7.1: Population by Year.

We can also plot two Series against each other (ignoring the index). In matplotlib:

```
>>> plt.plot(crime["Population"], crime["Burglary"])
>>> plt.show()
```

With `DataFrame.plot()`, specify the x-values as a keyword argument:

```
>>> crime.plot(x="Population", y="Burglary")
>>> plt.show()
```

Both procedures produce the same line plot, but the DataFrame method automatically sets the limits and labels of each axis and includes a legend. See Figure 7.2.

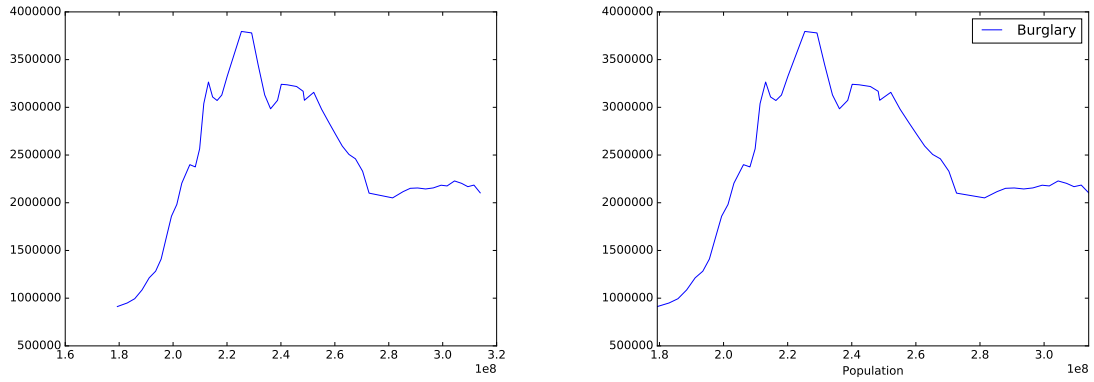


Figure 7.2: On the left, the result of `plt.plot()`. On the right, the result of `DataFrame.plot()`

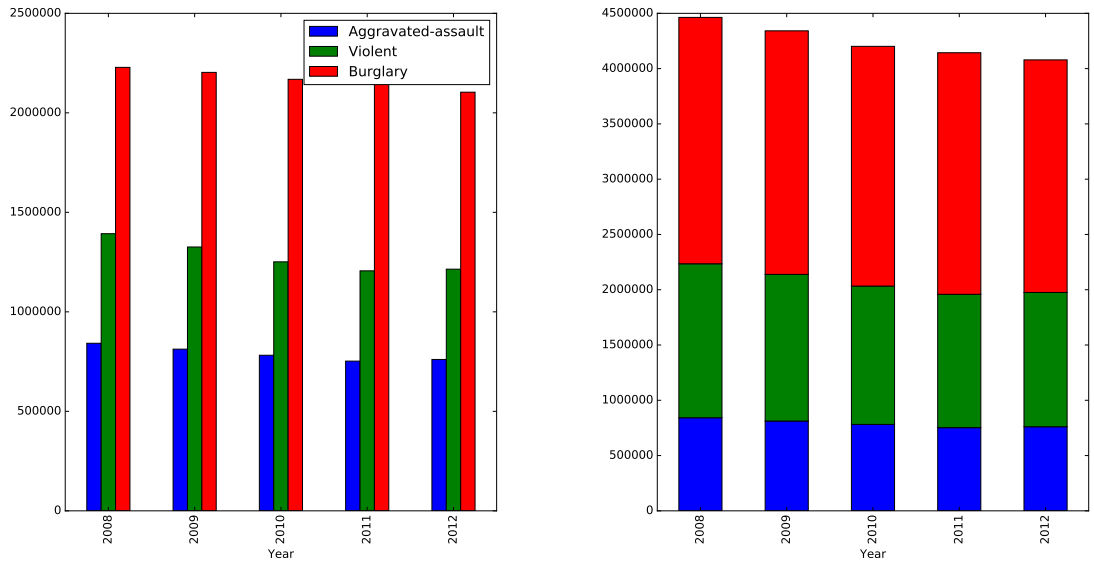
Standard matplotlib keyword arguments can be passed in as usual to `DataFrame.plot()`. This includes the ability to produce subplots quickly, modify the linestyle, and so on.

```
>>> crime.plot(subplots=True, layout=(4,3), linewidth=3, style="--", legend=↩
    False)
>>> plt.show()
```

Bar Plots

By default, the DataFrame's `plot()` function creates a line plot. We can create other types of plots easily by specifying the keyword `kind`. Bar plots are particularly useful for comparing several categories of data over time, or whenever there is a sense of progression in the index. Line plots are better suited to show a more continuous index (such as each year in a century), whereas bar plots are better for a discrete index (a few distinct years). Consider, for example, three different types of crime over the last five years. The argument `stacked` defaults to `False` (and `legend` to `True`). Stacking the bars can help to show the combined totals each year. Both types are shown below:

```
# Each call to plot() makes a separate figure automatically.
>>> crime.iloc[-5:][["Aggravated-assault", "Violent", "Burglary"]].plot(kind="↩
    bar")
>>> crime.iloc[-5:][["Aggravated-assault", "Violent", "Burglary"]].plot(kind="↩
    bar", stacked=True, legend=False)
>>> plt.show()
```



Problem 1. The `pydataset` module[a] contains numerous data sets stored as pandas DataFrames.

```
from pydataset import data
# "data" is a pandas DataFrame with IDs and descriptions.
# Call data() to see the entire list.
# To load a particular data set, enter its ID as an argument to data().
titanic_data = data("Titanic")
# To see the information about a data set, give data() the dataset_id with↩
    show_doc=True.
data("Titanic", show_doc=True)
```

Examine the data sets with the following `pydataset` IDs:

1. `nottem`: Average air temperatures at Nottingham Castle in Fahrenheit for 20 years.
2. `VADeaths`: Death rates per 1000 in Virginia in 1940.
3. `Arbuthnot`: Ratios of male to female births in London from 1629-1710.

Use line and bar plots to visualize each of these data sets. Decide which type of plot is more appropriate for each data set, and which columns to plot together. Write a short description of each data set based on the docstrings of the data and your visualizations.

[a] Run `pip install pydataset` if needed

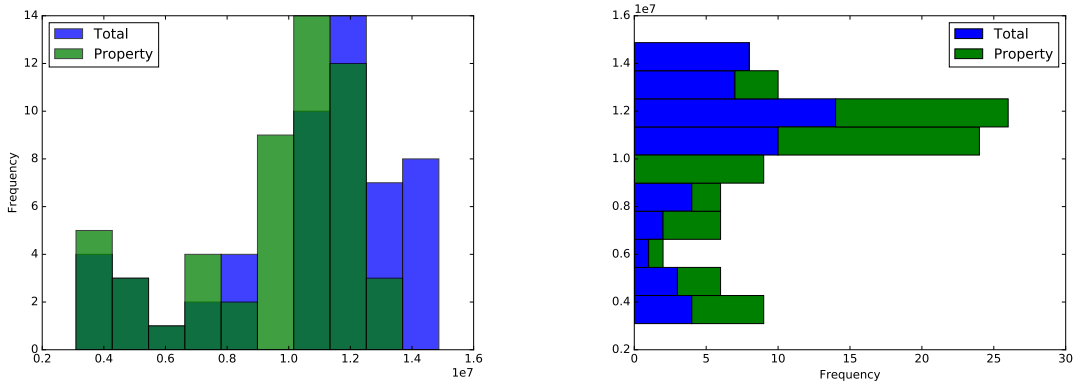
Histograms

Line and bar plots work well when there is a logical progression in the index, such as time. However, when frequency of occurence is more important than the location of the data, histograms and box plots can be more informative. Use `plot(kind="hist")` to produce a histogram. Standard histogram options, such as the number of bins, are also accepted as keyword arguments. The `alpha` keyword argument makes each bin slightly transparent.

```
>>> crime[["Total", "Property"]].plot(kind="hist", alpha=.75)
>>> plt.show()
```

Alternatively, the bins can be stacked on top of each other by setting the `stacked` keyword argument to `True`. We can also make the histogram horizontal by seting the keyword `orientation` to "horizontal".

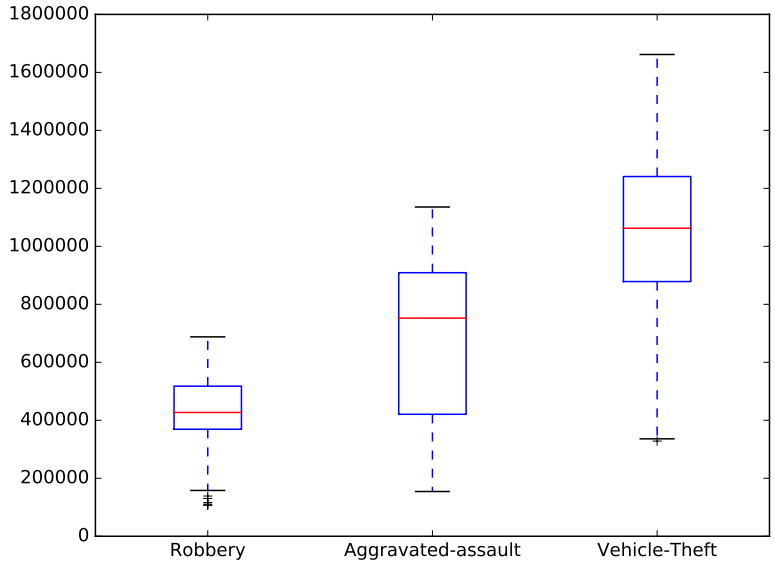
```
>>> crime[["Total", "Property"]].plot(kind="hist", stacked=True, orientation="↩
    horizontal")
>>> plt.show()
```



Box Plots

Sometimes it is helpful to visualize a distribution of values using the box-and-whisker plot which displays the median, first and third quartiles, and outliers. Like the previous examples, select the columns to examine and plot them with `plot()`. To switch the orientation, use `vert=False`.

```
crime[["Robbery", "Aggravated-assault", "Vehicle-Theft"]].plot(kind="box")
plt.show()
```



Problem 2. Examine the data sets with the following `pydataset` IDs:

1. `trees`: Girth, height and volume for black cherry trees.
2. `road`: Road accident deaths in the United States.
3. `birthdeathrates`: Birth and death rates by country.

Use histograms and box plots to visualize each of these data sets. Decide which type of plot is more appropriate for each data set, and which columns to plot together. Write a short description of each data set based on the docstrings of the data and your visualizations.

Scatter Plots

Scatter plots are commonly used in a myriad of areas and have a simple implementation in pandas. Unlike other plotting commands, `scatter` needs both an `x` and a `y` column as arguments.

The scatter plot option includes many features which can be used to make the plots easier to understand. For example, we can change the size of the point based on another column. Consider the pydataset `HairEyeColor`, which contains the hair and eye color of various individuals. A scatter plot of hair color vs eye color is relatively useless unless we can see the frequencies with which each combination occurs. Including the keyword argument `s` allows us to control the size of each point. This can be set to a fixed value or the value in another column. In the example below, the size of each point is set to the frequency with which each observation occurs.

```
>>> hec = data("HairEyeColor")
>>> X = np.unique(hec["Hair"], return_inverse=True)
>>> Y = np.unique(hec["Eye"], return_inverse=True)
```

```
>>> hec["Hair"] = X[1]
>>> hec["Eye"] = Y[1]
>>> hec.plot(kind="scatter", x="Hair", y="Eye", s=hec["Freq"]*20)
>>> plt.xticks([0,1,2,3], X[0])
>>> plt.yticks([0,1,2,3], Y[0])
>>> plt.show()
```

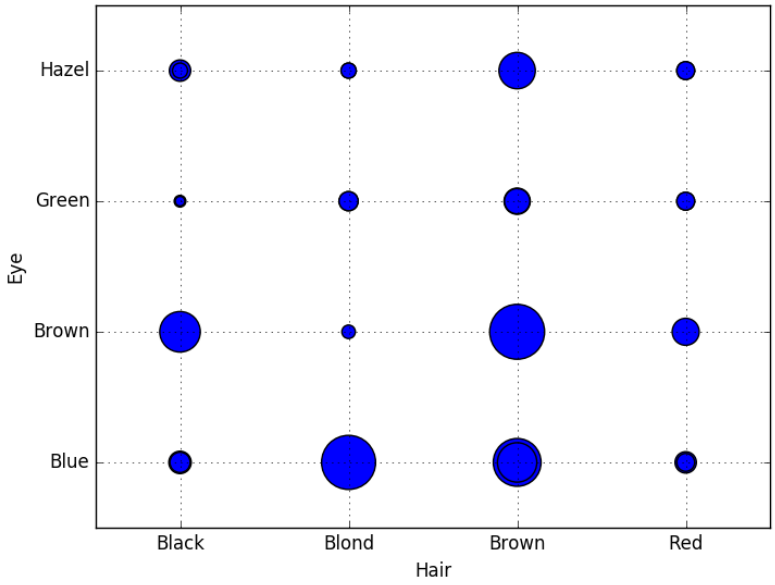


Figure 7.3: Frequency of Hair-Eye Color Combinations

Hexbins

While scatter plots are a great visualization tool, they can be uninformative for large datasets. It is nearly impossible to tell what is going on in a large scatter plot, and the visualization is therefore of little value. Hexbin plots solve this problem by plotting point density in hexagonal bins. With hexbins, the structure of the data is easy to see despite the noise that is still present. Following is an example using pydataset's `sat.act` plotting the SAT Quantitative score vs ACT score of students

```
>>> satact = data("sat.act")
>>> satact.plot(kind="scatter", x="ACT", y="SATQ")
>>> satact.plot(kind="Hexbin", x="ACT", y="SATQ", gridsize=20)
>>> plt.show()
```

Note the scatter plot in Figure 7.4. While we can see the structure of the data, it is not easy to differentiate the densities of the areas with many data points. Compare this now to the hexbin plot in the same figure. The two plots are clearly similar, but with the hexbin plot we have the added dimension of density.

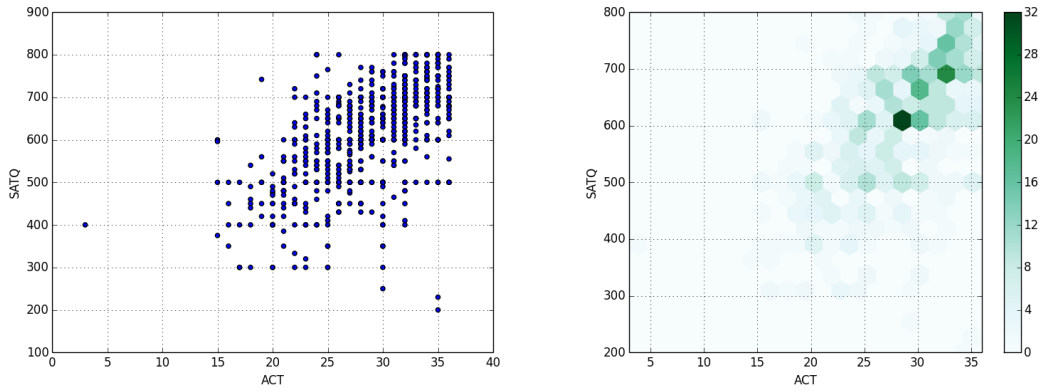


Figure 7.4: Comparing a scatter plot of SAT and ACT scores with a hexbin plot.

A key factor in creating an informative hexbin is choosing an appropriate `gridsize` parameter. This determines how large or small the bins will be. A large gridsize will give many small bins and a small gridsize gives a few large bins. Figure 7.5 shows the effect of changing the gridsize from 20 to 10.

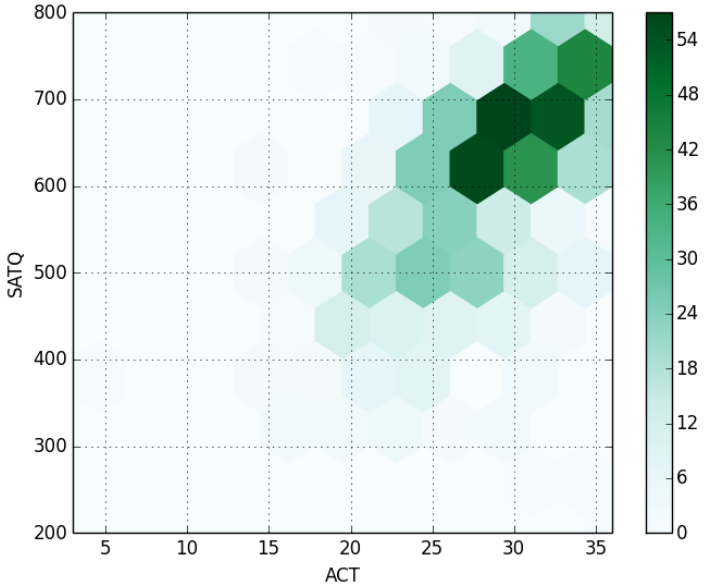


Figure 7.5: Hexbin Plot With Gridsize 10

Lag Plot

We are frequently interested in whether or not data which we have collected is random. Lag plots are used to investigate the randomness of a dataset. If the data is in fact random, then the lag plot will exhibit no structure, while nonrandom data will exhibit some kind of structure. Unfortunately, this does not give us an idea of what exactly that structure may be, but it is a quick and effective way to investigate the randomness of a given dataset.

```
>>> from pandas.tools.plotting import lag_plot
>>> randomdata = pd.Series(np.random.rand(1000))
>>> lag_plot(randomdata)
>>> plt.show()

>>> structureddata = pd.Series(np.sin(np.linspace(-10*np.pi, 10*np.pi, num=100)↩
    ))
>>> lag_plot(structureddata)
>>> plt.show()
```

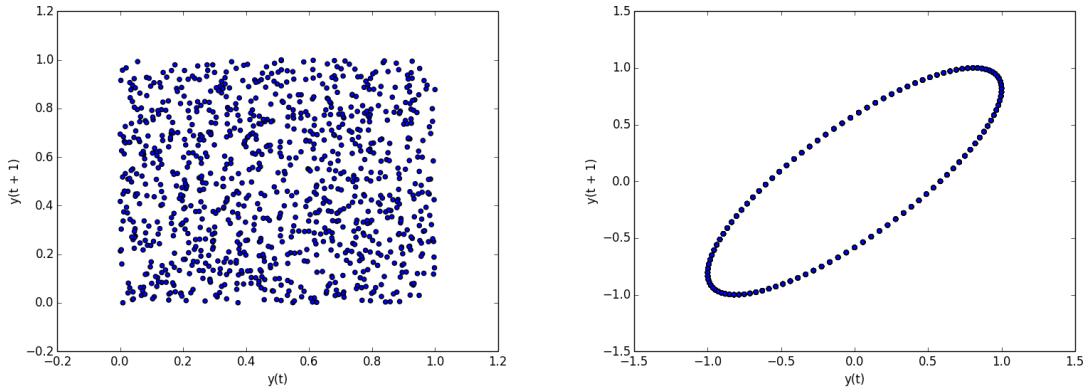


Figure 7.6: Lag plot of random data (left) compared to a lag plot of a sine wave (right).

Problem 3. Choose a dataset provided in the `pydataset` and produce scatter and hexbin plots demonstrating some characteristic of the data. A list of datasets in the `pydataset` module can be produced using the `data()` command.

For more types of plots available in Pandas and further examples, see `http://pandas.pydata.org/pandas-docs/stable/visualization.html`.

Data Visualization

Visualization is much more than a set of pretty pictures scattered throughout a paper for the sole purpose of providing contrast to the text. When properly implemented, 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.

Catching all of the Details

Consider the plot in Figure 7.7. What does it depict? We can tell from a simple glance that 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. We have not specified the units for the x or the y axis, we have no idea what `cons` is, there is no title, and we don't even know where the data came from in the first place.

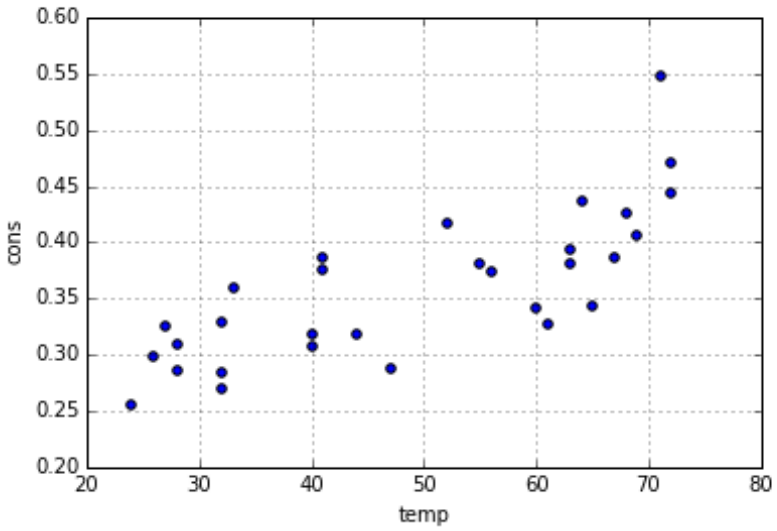


Figure 7.7: Non-specific data.

Labels, Legends, and Titles

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. Clearly, we need to explain our data in a useful manner that includes all of the vital information.

Consider again Figure 7.7. This figure comes from the `Icecream` dataset within the `pydataset` package, which we store here in a dataframe and then plot:

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

We have at this point reproduced the rather substandard plot in Figure 7.7. Using `data('Icecream', show_doc=True)` we find the following information:

1. The dataset details ice cream consumption via four-weekly observations from March 1951 to July 1953 in the United States.

2. `cons` corresponds to "consumption of ice cream per head" and is measured in pints.

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

4. The listed source is: "Hildreth, C. and J. Lu (1960) _Demand relations with autocorrelated disturbances_, Technical Bulletin No 2765, Michigan State University."

We add these important details using the following code. As we have seen in previous examples, pandas automatically generates legends when appropriate. However, although pandas also automatically labels the x and y axes, our data frame column titles may be insufficient. Appropriate titles for the x and y axes must also list appropriate units. For example, the y axis should specify that the consumption is in units of *pints per head*, in place of the ambigious label `cons`.

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

Unfortunately, there is no explicit function call that allows us to add our source information. To arbitrarily add the necessary text to the figure, we may use either `plt.annotate` or `plt.text`.

```
>>> plt.text(20, .1, "Source: Hildreth, C. and J. Lu (1960) _Demand relations ↩
    with autocorrelated disturbances_\nTechnical Bulletin No 2765, Michigan ↩
    State University.", fontsize=7)
```

Both of these methods are imperfect, however, and can normally be just as easily replaced by a caption attached to the figure in your presentation or document setting. We again 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 7.8.

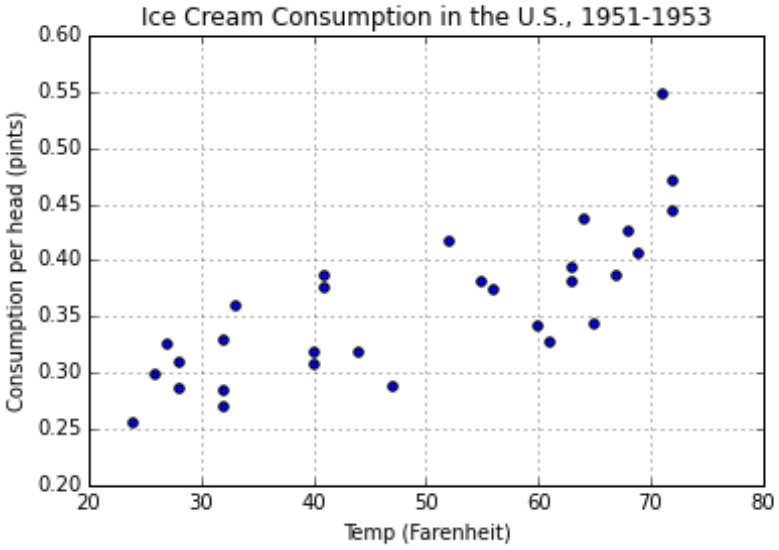


Figure 7.8: Source: Hildreth, C. and J. Lu (1960) _Demand relations with autocorrelated disturbances_, Technical Bulletin No 2765, Michigan State University.

Problem 4. Return to the plots you generated in problem 1 (datasets `nottem`, `VADeaths`, and `Arbuthnot`). Reproduce and modify these plots to include:

- A clear title, with relevant information for the period or region the data was collected in.
- Axes that specify units.
- A legend (for comparison data).
- The source. You may include the source information in your plot or print it after the plot at your discretion.

Note that in this problem, as well as for plots in subsequent labs, points will be taken off if any of these items are partially or fully missing from your graphs.

Choosing the Right Plot

Now that we know how to add the appropriate details for remaining plots, we return to the fundamental question of data visualization: Which plot should you use for your data? In previous sections, we have already discussed the various strengths and weaknesses of available pandas plotting techniques. At this point, we know how to visualize data using line graphs, bar charts, histograms, box plots, scatter plots, hexbins, and lag plots.

However, perfectly plot-ready data sets–organized by a simple continuum or a convenient discrete set–are few and far between. In the real world, it is rare to find data that is perfectly ready to become a meaningful visual, even if it is already a `DataFrame`. As such, deciding how to organize and group your data is one of the most important parts of "choosing the right plot".

In Lab 8 we will go over how to group data in meaningful ways. You should then be better able to choose the right type of plot for your data.

8 Pandas III: Grouping and Presenting Data

Lab Objective: *Learn about Pivot tables, groupby, etc.*

Introduction

Pandas originated as a wrapper for numpy that was developed for purposes of data analysis. Data seldom comes in a format that is perfectly ready to use. We always need to be able to interpret what our data is telling us. Some data may be of little or no use, while other aspects of our data may be vital. *Pivoting* is an extremely useful way to sort through data and be able to present results clearly and compactly. Two central ways to accomplish this are by using `groupby` and by using *Pivot Tables*.

Groupby

In Lab 7 we introduced `pydatasets`, and mentioned how, in their raw format, plotting would be nonsensical. Many datasets are simply composed of tables of individuals (represented as rows), with a list of classifiers associated with each one (columns).

For example, consider the `msleep` dataset. In order to view the columns present in this dataset, we make use of the function `head()`. This will show us the first five rows, and all of the columns.

```
>>> import pandas as pd
>>> from pydataset import data
>>> msleep = data("msleep")
>>> msleep.head()
                        name       genus   vore         order  conservation
1                    Cheetah    Acinonyx  carni     Carnivora            lc
2                  Owl monkey       Aotus   omni      Primates           NaN
3             Mountain beaver  Aplodontia  herbi      Rodentia            nt
4  Greater short-tailed shrew     Blarina   omni  Soricomorpha            lc
5                        Cow         Bos  herbi  Artiodactyla  domesticated

   sleep_total  sleep_rem  sleep_cycle  awake  brainwt   bodywt
1         12.1        NaN          NaN   11.9      NaN   50.000
2         17.0        1.8          NaN    7.0  0.01550    0.480
3         14.4        2.4          NaN    9.6      NaN    1.350
```

```
4          14.9         2.3      0.133333     9.1   0.00029     0.019
5           4.0         0.7      0.666667    20.0   0.42300   600.000
```

Each row consists of a single type of mammal and its corresponding identifiers, including genus and order, as well as sleep measurements such as total amount of sleep (in hours) and REM sleep, in hours. When we try to plot this data using plt.plot, the individual data points do not demonstrate overall trends.

```
>>> msleep.plot(y="sleep_total", title="Mammalian Sleep Data", legend=False)
>>> plt.xlabel("Animal Index")
>>> plt.ylabel("Sleep in Hours")
>>> plt.show()
```

The above code results in Figure 8.1.

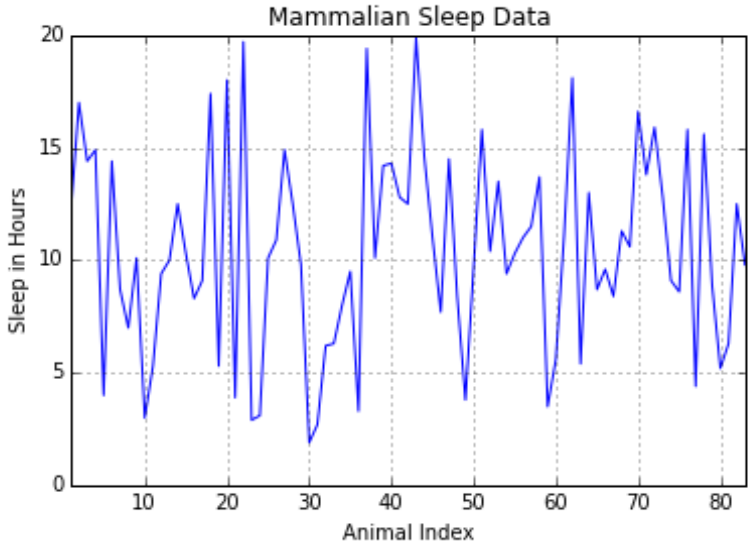


Figure 8.1: Source: Proceedings of the National Academy of Sciences, 104 (3):1051-1056, 2007. Updates from V. M. Savage and G. B. West, with additional variables supplemented by Wikipedia.

This set of connected data points is not particularly revealing, as it plots the first numerical column, `sleep_total`, as a function of the animal index, which is seemingly random.

The `DataFrame` contains information that will help us to make better sense of this data. Using `groupby()`, we can select the parts we want. As the name implies, `groupby()` takes a `DataFrame` and creates different groupings. For this dataset, let's consider the sleep differences between herbivores, omnivores, insectivores, and carnivores. To do so, we simply call the `groupby` method on the `vore` column to obtain a `groupby` object organized by diet classification.

```
>>> vore = msleep.groupby("vore")
```

You can also group the data by multiple columns, for example, both the `vore` and `order` classifications. To group and view this data, simply use:

```
>>> vorder = msleep.groupby(["vore", "order"])

# View groups within vorder
>>> vorder.describe()
```

This listing is much too long to view here, but you should be able to see that it first sorts all rows into the appropriate `vore` category, and then into the appropriate `order` category. It gives the count of rows in each section, along with the mean, standard deviation, and other potentially useful statistics.

Pandas `groupby` objects are not lists of new `DataFrames` associated with groupings. They are better thought of as a dictionary or generator-like object which can be *used* to produce the necessary groups. However, the `get_group()` method will do this, as follows:

```
# Get carnivore group
>>> Carni = vore.get_group("carni")
# Get herbivore group
>>> Herbi = vore.get_group("herbi")
```

The `groupby` object includes many useful methods that can help us make visual comparisons between groups. The `mean()` method, for example, returns a new `DataFrame` consisting of the mean values attached to each group. Using this method on our `vore` object returns a nicely organized `DataFrame` of the average sleep data for each mammalian diet pattern. We can similarly create a `DataFrame` of the standard deviations for each group.

At this point, we have a nicely organized dataset that can easily be turned into a bar chart. Here, we use the `DataFrame.loc` method to access three specific columns in the bar chart (`sleep_total`, `sleep_rem`, and `sleep_cycle`).

```
>>> means = vore.mean()
>>> errors = vore.std()
>>> means.loc[:,["sleep_total", "sleep_rem", "sleep_cycle"]].plot(kind="bar", ↩
    yerr=errors, title="Mean Mammallian Sleep Data")
>>> plt.xlabel("Mammal diet classification (vore)")
>>> plt.ylabel("Hours")
>>> plt.show()
```

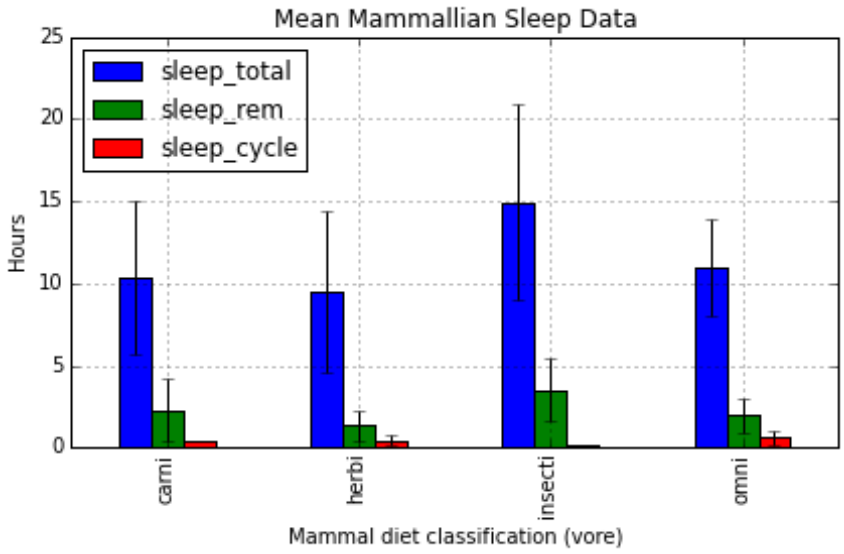


Figure 8.2: Source: Proceedings of the National Academy of Sciences, 104 (3):1051-1056, 2007. Updates from V. M. Savage and G. B. West, with additional variables supplemented by Wikipedia.

Problem 1. Examine the `diamonds` dataset found in the `pydataset` module. This dataset contains the identifiers and attributes of 53,940 individual round cut diamonds. Using the `groupby` method, create a visual highlighting and comparing different aspects of the data. This can be in the form of a single plot or comparative subplots. Use the plotting techniques from Lab 7.

Print a paragraph explaining what type of graph you used, why, and what we learn about the dataset from your plot. Don't forget to include titles, clear labels, and sourcing.

The following is an example of comparative subplots, which may *not* be used for your plot.

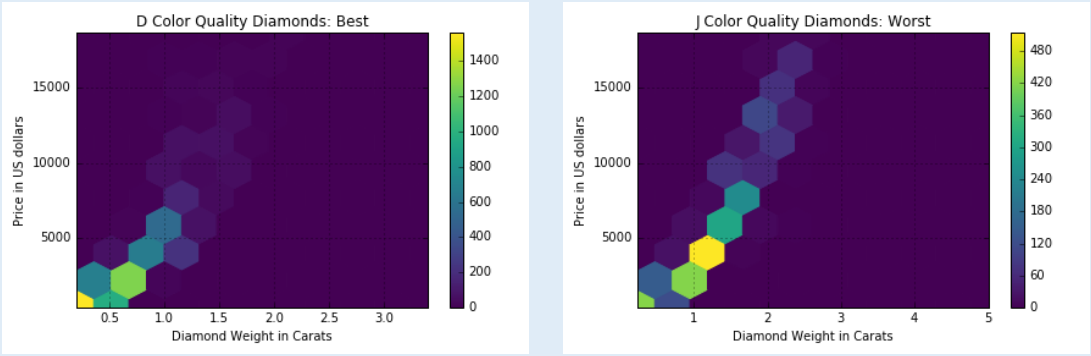


Figure 8.3: Source: Adopted from R Documentation

The following is an appropriate (if lengthy) description of our plots:

The above plots were created using `groupby` on the diamond colors and then using a hexbin comparing carats to price for the highest and lowest quality diamonds, respectively. This hexbin is particularly revealing for each set of thousands of diamonds because it meaningfully displays concentration of datapoints. Matplotlib's new `viridis` colorplot, with a dark background, reveals bins that would have been invisible with a white background. By comparing these plots, we note that the greatest number of J quality diamonds in the dataset are about 1.25 carats and $4000 dollars in price, whereas the highest concentration of D quality diamonds are smaller and therefore cheaper. We may attribute this to D quality diamonds being rarer, but the colorbar on the side reveals that D diamond numbers are, in fact, far higher than those of the J color. Instead it is simply more likely that D quality diamonds of larger sizes are rarer than those of smaller sizes. Both hexbins reveal a linearity between diamond weight and diamond price, with D diamonds showing more variability and J diamonds displaying a strict linearity.

`Groupby` is a very useful method to order data. However, it is not the perfect tool for every situation. As we saw when sorting data by multiple columns, we can end up with a useful grouping, but too much information to display. If we want to see information in table format, we can make use Pivot Tables.

Pivot Tables

With a given pandas `DataFrame`, we can visualize data easily with the method `pivot_table()`. The Titanic dataset (`titanic.csv`) is especially apt for this. It includes many columns, but we will use only `Survived`, `Pclass`, `Sex`, `Age`, `Fare`, and `Embarked` here. You will need to load in this dataset for upcoming problems, using principles taught in Lab 6. Note that many of the ages are missing values. For our purposes, simply fill these missing values with the average age. Then drop any rows that are missing data. Once we have a usable dataset, we can make Pivot Tables to view trends in the data.

```
>>> titanic.pivot_table('Survived', index='Sex', columns='Pclass')
Pclass          1         2         3
Sex
female   0.962406  0.893204  0.473684
male     0.350993  0.145570  0.169540
```

This simple command makes a table with rows `Sex` and columns `Pclass`, and averages the result of the column `Survived`, thereby giving the percentage of survivors in each grouping. Note that this is similar to `groupby`: it sorts all entries into their gender and class, and the "function" it performs is getting the percentage of survivors.

This is interesting, and we can clearly see how much more likely females were to survive than males. But how does age factor into survival rates? Were male children really that likely to die as compared to females in general?

We can investigate these results by *multi-indexing*. We can pivot based on more than just two variables, by adding in another index. Note that in the original dataset, the column `Age` has a floating point value for the age of each passenger. If we were to just add 'age' as an argument for index, then the table would create a new row for *each* age present. This wouldn't be a very useful or simple table to visualize, so we desire to partition the ages into 3 categories. We use the function `cut()` to do this.

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

Now with this other partition of the column age, we can add this dimension to our pivot table, passing it along with `Sex` in a list to the parameter `index`.

```
>>> # Add a third dimension, age, to our pivot table
>>> titanic.pivot_table('Survived', index =['Sex', age], columns='Pclass')
Pclass                  1         2         3
Sex    Age
female (0, 12]   0.000000  1.000000  0.466667
       (12, 18]  1.000000  0.875000  0.607143
       (18, 80]  0.966667  0.878049  0.436170
male   (0, 12]   1.000000  1.000000  0.342857
       (12, 18]  0.500000  0.000000  0.081081
       (18, 80]  0.328671  0.087591  0.159420
```

What do we notice? First of all, 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. However, look at the female children in first class. It says that zero percent survived. This might seem a little odd, but if we looked at our data set again to see how many passengers fell into this category of female, 1st class, and age 0 to 12, we would find only one passenger. Therefore, the statistic that 0% of female children in first class lived is misleading. We can see the number of people in each category by simply specifying the parameter aggfunc=‘count’.

```
>>> titanic.pivot_table('Survived', index=['Sex',age], columns='Pclass', ↩
    aggfunc='count')
Pclass             1    2    3
Sex    Age
female (0, 12]     1   13   30
       (12, 18]   12    8   28
       (18, 80]  120   82   94
male   (0, 12]     4   11   35
       (12, 18]    4   10   37
       (18, 80]  143  137  276
```

The parameter aggfunc defaults to ‘mean’, which is why we have seen the mean survival rate for each of the different categories. By specifying aggfunc to be ‘count’, we get how many passengers of each category are present in the table. In this case, specifying the value is ‘survived’ is redundant. We get the same table if we put in `Fare` in place of `Survived`.

This table brings up another point about partitioning datasets. This Titanic dataset includes data for only about 1000 passengers. This is not that large of a sample size, relatively speaking, and significant sample sizes aren’t guaranteed for each possible partitioning of the the data columns. It is a good practice to ask questions about the numbers you see in pivot tables before making any conclusions.

Now, for fun, let’s add a fourth dimension to our table–the cost of the passenger’s ticket. Let’s add this dimension to our columns of the pivot table. We now use the function `qcut()` to partition the fare column’s data into two different categories.

```
>>> # Partition fare column into 2 categories based on the values present in ↩
    fare column
>>> fare = pd.qcut(titanic['fare'], 2)
>>> # Add the fare as a dimension of columns
>>> titanic.pivot_table('Survived', index=['Sex',age], columns=[fare, 'Pclass'↩
    ])
Fare                        [0, 15.75]                    (15.75, 512.329]
Pclass               1        2         3          1           2          3
Sex        Age
female   (0, 12]    NaN   1.000000  0.600000    0.000000    1.000000    ↩
    0.400000
         (12, 18]   NaN   0.750000  0.666667    1.000000    1.000000    ↩
            0.250000
         (18, 80]   NaN   0.875000  0.434783    0.966667    0.880000    ↩
            0.440000
male     (0, 12]    NaN   1.000000  0.636364    1.000000    1.000000    ↩
    0.208333
         (12, 18]   NaN   0.000000  0.115385    0.500000    0.000000    ↩
            0.000000
         (18, 80]   0.2   0.113636  0.153846    0.333333    0.040816    ↩
            0.206897
```

We now see something else about our dataset–we get NaNs in some entries. The dataset has some invalid entries or incomplete data for the column fare. Let's investigate further.

By inspecting our dataset, we can see that there might not be any people in the categories given in the pivot table. We can fill in the NaN values in the pivot table with the argument `fill_value`. We show this below.

```
>>> # Specify fill_value = 0.0
>>> titanic.pivot_table('survived', index=['sex',age], columns=[fare, 'class'],↩
     fill_value=0.0)
Fare                        [0, 15.75]                    (15.75, 512.329]
Pclass               1        2         3          1           2          3
Sex        Age
female   (0, 12]    0.0   1.000000  0.600000    0.000000    1.000000    ↩
    0.400000
         (12, 18]   0.0   0.750000  0.666667    1.000000    1.000000    ↩
            0.250000
         (18, 80]   0.0   0.875000  0.434783    0.966667    0.880000    ↩
            0.440000
male     (0, 12]    0.0   1.000000  0.636364    1.000000    1.000000    ↩
    0.208333
         (12, 18]   0.0   0.000000  0.115385    0.500000    0.000000    ↩
            0.000000
         (18, 80]   0.2   0.113636  0.153846    0.333333    0.040816    ↩
            0.206897
```

It should be noted though that with datasets where NaN's occur frequently, some preprocessing needs to be done so as to get the most accurate statistics and insights about your dataset. You should've worked on cleaning datasets in previous lab, so we don't go further into detail about the topic here.

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

1. Using a `groupby()` call, see what the survival rates of the passengers were based on where they embarked from (`embark_town`).
2. Next, create a pivot table to further look at survival rates based on both place of embarkment and gender.
3. What do these tables suggest to you about the significance of where people embarked in influencing their survival rate? Examine the context of the problem, and explain what you think this really means. Find 2 more pivot tables that investigate the claim further, looking at other criterion (e.g., class, age, etc.). Include explanations.

As you have learned in the pandas labs, pandas is a powerful tool for data processing, and has a plethora of built-in functions that greatly simplify data analysis. Ler's now put the skills you have acquired to practical use.

Problem 3. Search through the `pydatasets` and find one which you would like to present. Make sure there is enough data to produce a proper presentation. Process the data appropriately, and create your own presentation, as though you were presenting to a superior in the appropriate line of work. Your presentation can be in any format you like, but as a bare minimum, it must include the following:

- An appropriate title.
- 3 charts or visuals with captions.
- 3 tables with captions.
- 2 paragraps of textual explanations.
- Appropriate data sourcing.

9 Pandas IV: Time Series

Lab Objective: *Learn how to manipulate and prepare time series in pandas in preparation for analysis*

Introduction: What is time series data?

Time series data is ubiquitous in the real world. Time series data is any form of data that comes attached to a timestamp (i.e. Sept 28, 2016 20:32:24) or a period of time (i.e. Q3 2012). Some examples of time series data include:

- stock market data
- ocean tide levels
- number of sales over a period of time
- website traffic
- concentrations of a certain compound in a solution over time
- audio signals
- seismograph data
- and more...

Notice that a common feature of all these types of data is that the values can be tied to a specific time or period of time.

In this lab, we will not go into depth on the analysis of such data. Rather, we will discuss some of the tools provided in pandas for cleaning and preparing time series data for further analysis.

Initializing Time Series in `pandas`

To take advantage of all the time series tools available to us in pandas, we need to make a few adjustments to our normal DataFrame.

The `datetime` Module and Initializing a `DatetimeIndex`

For pandas to know to treat a DataFrame or Series object as time series data, the index must be a `DatetimeIndex`. pandas utilizes the `datetime.datetime` object from the `datetime` module to standardize the format in which dates or timestamps are represented.

```
>>> from datetime import datetime

>>> datetime(2016, 9, 28) # 9/28/2016
datetime.datetime(2016,9,28,0,0)

>>> datetime(2016, 9, 28, 21, 12, 48) # 9/28/2016 9:12:48 PM
datetime.datetime(2016, 9, 28, 21, 12, 48)
```

Unsurprisingly, the format for dates varies greatly from dataset to dataset. The `datetime` module comes with a string parser (`datetime.strptime()`) flexible enough to translate nearly any format into a `datetime.datetime` object. This method accepts the string representation of the date, and a string representing the format of the string. See Table 9.1 for all the options.

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

Table 9.1: Formats recognized by `datetime.strptime()`

Here are some examples of using `datetime.strptime()` to parse the same date from different formats.

```
>>> datetime.strptime("2016-9-28", "%Y-%m-%d")
datetime.datetime(2016, 9, 2, 0, 0)

>>> datetime.strptime("9/28/16", "%m/%d/%y")
datetime.datetime(2016, 9, 2, 0, 0)

>>> datetime.strptime("2016-9-28 9:12:48", "%Y-%m-%d %I:%M:%S")
datetime.datetime(2016, 9, 28, 9, 12, 48)
```

If the dates are in an easily parsible format, pandas has a method `pd.to_datetime()` that can turn a whole pandas `Series` into `datetime.datetime` objects. In the case of the index, the index is automatically converted to a `DatetimeIndex`. This index type is what distinguishes a regular `Series` or `DataFrame` from a time series.

```
>>> dates = ["2010-1-1", "2010-2-1", "2010-3-1", "2011-1-1",
... "2012-1-1", "2012-1-2", "2012-1-3"]
```

```
>>> values = np.random.randn(7,2)
>>> df = pd.DataFrame(values, index=dates)
>>> df
                 0         1
2010-1-1  0.566694  1.093125
2010-2-1 -0.219856  0.852917
2010-3-1  1.511347 -1.324036
2011-1-1  0.300766  0.934895
2012-1-1  0.212141  0.859555
2012-1-2  1.483123 -0.520873
2012-1-3  1.436843  0.596143

>>> df.index = pd.to_datetime(df.index)
```

> **Note**
>
> In earlier versions of pandas, there was a dedicated TimeSeries data type. This has since been depricated, however the functionality remains. Therefore, if you happen to read any materials that reference the TimeSeries data type, know that the cooresponding functionality is likely still in place as long as you have a `DatetimeIndex` associated with your Series or DataFrame.

Problem 1. The provided dataset, "DJIA.csv" is the daily closing value of the Dow Jones Industrial Average for every day over the past 10 years. Read this dataset into a Series with a DatetimeIndex. Replace any missing values with `np.nan`. Lastly, cast all the values in the Series to floats. We will use this dataset for many problems in this lab.

Handling Data Without Marked Timestamps

There will be times you will need to analyze time series data that does not come marked with an index. For example, you may have the a list of bank account balances at the beginning of every month for the last 5 years. You may have heart rate readings every 10 minutes for the past week. pandas provides efficient tools for generating indices for these kinds of siutations.

The `pd.date_range()` Method

The `pd.date_range()` method is analogous to `np.arange()`. The parameters we will use most are described in Table 9.2.

Exactly two of the parameters `start`, `end`, and `periods` must be defined to generate a range of dates. The `freqs` parameter accepts a variety of string representations. The accepted strings are referred to as *offset aliases* in the documentation. See Table 9.3 for a sampling of some of the options. For a complete list of the options, see `http://pandas.pydata.org/pandas-docs/stable/timeseries.html#offset-aliases`.

```
>>> pd.date_range(start='9/28/2016 16:00', periods=5)
```

`start`	start of date range
`end`	end of date range
`periods`	the number of dates to include in the date range
`freq`	the amount of time between dates (similar to “step")
`normalize`	trim the time of the date to midnight

Table 9.2: Parameters for `datetime.strptime()`

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

Table 9.3: Parameters for `datetime.strptime()`

```
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')

>>> 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')


>>> pd.date_range(start='9/28/2016 16:00',
            end='9/29/2016 16:30', freq="10T")
DatetimeIndex(['2016-09-28 16:00:00', '2016-09-28 16:10:00',
               '2016-09-28 16:20:00', '2016-09-28 16:30:00'],
              dtype='datetime64[ns]', freq='10T')
```

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

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

Problem 2. The "paychecks.csv" dataset has values of an hourly employee's last 93 paychecks. He started working March 13, 2008. This company hands out paychecks on the first and third Fridays of the month. However, "paychecks.csv" is not indexed explicitly as such. To be able to manipulate it as a time series in pandas, we will need to add a `DatetimeIndex` to it. Read in the data and use `pd.date_range()` to generate the `DatetimeIndex`.

Hint: to combine two `DatetimeIndex` objects, you can use the `.union()` method of `DatetimeIndex` objects.

Plotting Time Series

The process for plotting a time series is identical to plotting any other Series or DataFrame. For more information and examples, refer back to Lab 7.

Problem 3. Plot the DJIA dataset that you read in as part of Problem 1. Label your axes and title the plot.

Dealing with Periods Instead of Timestamps

It is often important to distinguish whether a given figure corresponds to a single point in time or to a whole month, quater, year, decade, etc. A `Period` object is better suited for the summary over a *period* of time rather than the timestamp of a specific event.

Some example of time series that would merit the use of periods would include,

- The number of steps a given person walks in a day.
- The box office results per week for a summer blockbuster.
- The population changes of a given city per year.
- etc.

The `Period` Object

The principle parameters of the `Period` are `"value"` and `"freq"`. The `"value"` paramter indicates the label for a given `Period`. This label is tied to the *end* of the defined `Period`. The `"freq"` indicates the length of the `Period` and also (in some cases) indicates the offset of the `Period`. The `"freq"` parameter accepts the majority, but not all, of frequencies listed in Table 9.3.

These nuances are best clarified through examples.

```
# The default value for `freq' is "M" meaning month.
>>> p1 = pd.Period("2016-10")
>>> p1.start_time
Timestamp('2016-10-01 00:00:00')

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

```
# A `freq' value of 'A-DEC' indicates a annual period
#    with the end of the period occuring in December.
>>> p2 = pd.Period("2016-10-03", freq="A-DEC")
>>> p2.start_time
Timestamp('2007-01-01 00:00:00')

>>> p2.end_time
Timestamp('2007-12-31 23:59:59.999999999')

# Notice that the Period object that is created is the
#   week of the year when the given date occurs. Also
#   note that "W-SAT" indicates we wish to treat
#   Saturday as the last day of the week (and therefore
#   Sunday as the first day of the week.)
>>> p3 = pd.Period("2016-10-03", freq="W-SAT")
>>> p3
Period('2016-10-02/2016-10-08', 'W-SAT')
```

The `pd.period_range()` Method

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.

```
# a 'freq' value of 'Q-DEC' means that Q4 ends in December.
>>> pd.period_range("2008", "2010-12", freq="Q-DEC")
PeriodIndex(['2009Q1', '2009Q2', '2009Q3', '2009Q4', '2010Q1', '2010Q2',
             '2010Q3', '2010Q4'], dtype='int64', freq='Q-DEC')
```

Other Useful Functionality and Methods

After creating a `PeriodIndex`, you have the option to redefine the `"freq"` parameter using the `asfreq()` method.

```
>>> p = pd.period_range("2010-03", "2011", freq="3M")
PeriodIndex(['2010-03', '2010-06', '2010-09', '2010-12'],
            dtype='int64', freq='3M')

# Change frequency to be "Q-DEC" instead of "3M"
>>> p = p.asfreq("Q-DEC")
>>> p
PeriodIndex(['2010Q2', '2010Q3', '2010Q4', '2011Q1'],
            dtype='int64', freq='Q-DEC')
```

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

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

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

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

Similarly, you can switch from timestamps to periods.

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

Problem 4. The “finances.csv" dataset has a list of simulated quarterly earnings and expense totals from a fictional company. Load these values into a DataFrame with a PeriodIndex with a quarterly frequency. Assume the fiscal year starts at the beginning of September and that the dataset goes back to September 1978.

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 a Time Series

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

The two main forms of resampling are downsampling (aggregating data into fewer intervals) and upsampling (adding more intervals). We will only address downsampling in detail.

Downsampling

When downsampling data, we aggregate our time series data into fewer intervals. Let's futher consider the website traffic examples given above.

Say we have a data set of website traffic indexed by timestamp. Each entry in the dataset contains the IP address of the user, the time the user entered the website, and the time the user left the website.

If we were interested in having information regarding the average visit duration for any given day, we could downsample the data to a daily timescale. To aggregate the data, we would take the average across all the datapoints for the day.

Instead, if we were interested in the number of visits to the website per hour, we could downsample to a weekly timescale. To aggregate the data in this case, we would count up the number of visits in a hour.

The task of downsampling is handled using the `resample()` method. The parameters we will use are `rule`, `how`, `closed`, and `label`. For explanations of these parameters, see Table 9.4.

Problem 5. Using the "website_traffic.csv" dataset, solve the problem described above. Namely, downsample the dataset to show daily average visit duration. In a different DataFrame, also downsample the dataset to show total number of visits per hour. Your resulting time series should use a `PeriodIndex`.

`rule`	the offset string (see Table 9.3)
`how`	the data aggregation method (i.e. “sum” or “mean”)
`closed`	the boundary of the interval that is closed/inclusive (default “right")
`label`	the boundary of the interval that is assigned to the label (default “right")

Table 9.4: Parameters for `DataFrame.resample()`

Basic Time Series Analysis

As mentioned in the beginning of this lab, the focus of this lab is not meant to be on Time Series Analysis. However, we would like to address a few very basic methods that are built into pandas.

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. See the following example code for clarification:

```
>>> 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 has a particularly useful application. We can easily 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)
               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. From the Dow Jones Industrial Average dataset referenced in Problem 1, use shifting to find the following information:

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

Hint: Downsample or use `groupby()` to answer the questions regarding months.

Rolling Functions and Exponentially-Weighted Moving Functions

In many situations, your data will be noisy. You will often be more interested in general trends. We can accomplish this through *rolling functions* and *exponentally-weighted moving (EWM)* functions.

Rolling functions, or intuitively called *moving window functions*, perform some kind of calculation on just a window of data. There are a few rolling functions that come standard with pandas. However, we will only cover a few of these. To visualize the effects these operations have on our time series, we will plot the results.

We first begin by defining the time series we will use throughout the remaining example code.

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

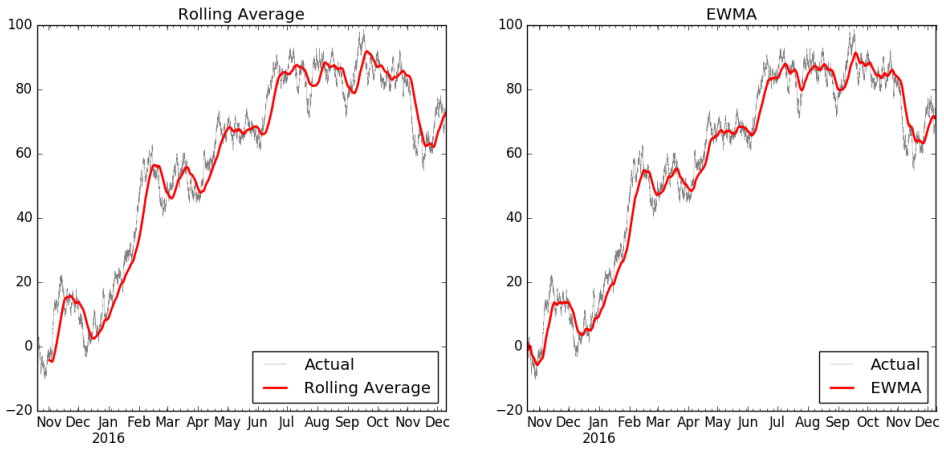


Figure 9.1: Rolling average and EWMA

Rolling Functions (Moving Window Functions)

One of the most commonly used rolling functions is the rolling average.

```
s.plot(lw=.3, color='grey', label="Actual")
s.rolling(window=200).mean().plot(color='r', lw=2, label="Moving Average")
plt.legend(loc="lower right")
plt.show()
```

Now let's break apart the syntax. The function call, `s.rolling(window=200)` creates a `pd.core.rolling.Window` object. You can then call any functions you would traditionally call on a DataFrame. For example, you can call `mean()`, `std()`, `var()`, `min()`, `max()`, `etc.`. As your intuition would suggest, rather than executing these functions across the whole time series, these functions are executed across each window and then aggregated into a new time series object.

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,

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

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.

```
s.plot(lw=.3, color='grey', label="Actual")
s.ewm(span=200).mean().plot(color='r', lw=2, label="EWMA")
plt.legend(loc="lower right")
plt.show()
```

Problem 7. In this problem, we will explore the differences between rolling functions and EWM functions.

Generate a time series plot with the following information from the DJIA dataset:

- The original data points.
- Rolling average with window 30.
- Rolling average with window 365.
- Exponential average with span 30.
- Exponential average with span 365.

Include a legend, axis labels, and title. Your plot should look like Figure **??**.

Problem 8. Plot the rolling minimum and rolling maximum values over the DJIA dataset with a window size of 30. This will create a moving bound for all the data that is easy to visualize.

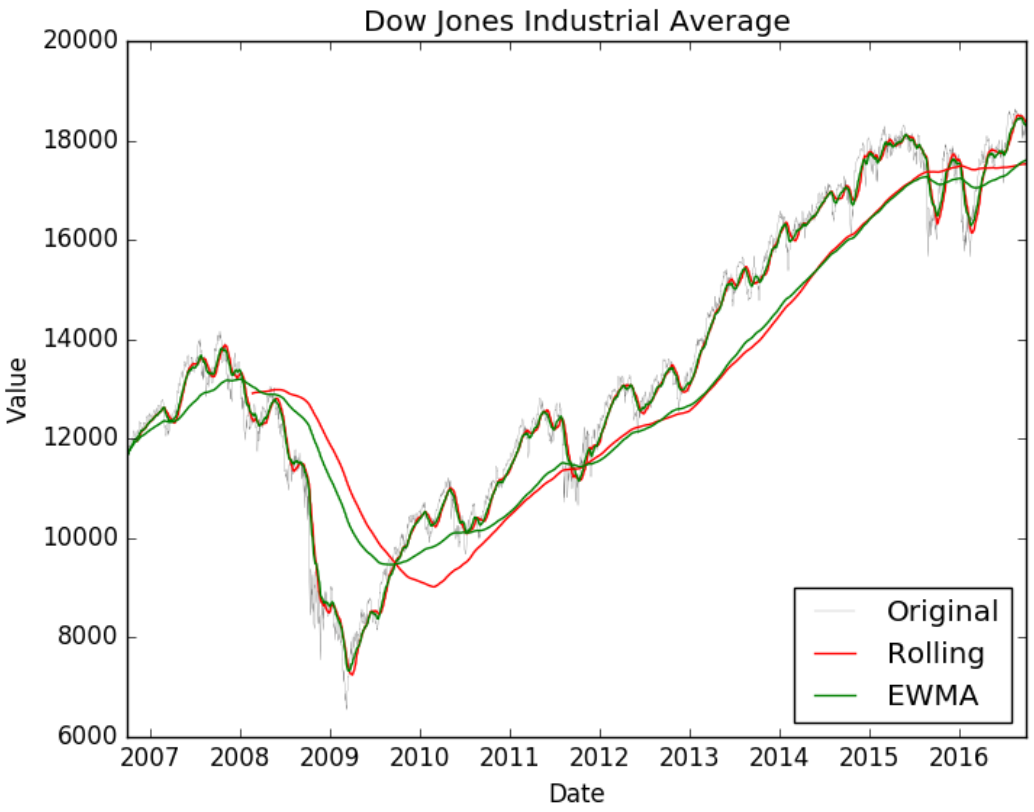


Figure 9.2: Rolling average and EWMA with windows 30 and 365.

10 Introduction to Bokeh

Lab Objective: *In this lab, we will introduce the Bokeh python package. We will use Bokeh to produce interactive, dependency-free data visualizations that can be viewed in any web browser.*

> ACHTUNG!
>
> This lab uses Bokeh version 0.12.0. It was released in the summer 2016. Bokeh is still under development, so the syntax is subject to change. However, the development is far enough along that the general framework has been solidified.

Bokeh is a visualization package focused on making plots that can be viewed and shared in web browsers. We have already addressed data visualization practices in previous labs. In this lab, we will not exhaustively address how to generate all the plots you have made using matplotlib. Rather, we will highlight some of the key differences in Bokeh. For all other questions not addressed in this lab, we direct the reader to the online Bokeh documentation, `http://bokeh.pydata.org/en/0.11.1/`.

Interactive Visualizations with Bokeh

One of the major selling points for the Bokeh Python package is the ability to generate interactive plots that can be viewed a web browser. The Bokeh Python package is being developed along with a Javascript library called BokehJS. The Bokeh Python package is simply a wrapper for this library. The BokehJS library handles all the visualizations in the web browser.

Throughout this lab, all the exercises will piece together to form one final web-based visualization. To see the final result, go to `THE_FINAL_PROJECT_HOSTED_ON_ACME/data_or_something`.

Even though there are many options for adding interactions to your plots, (see `http://bokeh.pydata.org/en/latest/docs/user_guide/interaction.html`) we will only focus on pan, zoom, hover, and the slider widget.

Cleaning the Data

For this project, we will be using the FARS (Fatality Analysis Reporting System) dataset. This is an incredibly rich data set of all fatal car accidents in the United States in a given year. We will be examining the data provided from 2010-2014. The data for these years are spread across several files.

The Accidents File

The columns found in the accidents table contains most of the data we will be analyzing. For the analysis we will be doing, we will be interested in the following columns:

- ST_CASE : The unique case ID. This will be used primarily for merging information from the other tables.
- STATE : State in which the accident occurred. There is a table at the end of this lab containing the relationship between IDs and states.
- LATITUDE : The latitude of the location where the accident occurred. Values of 77.7777, 88.8888, 99.9999 will be considered null.
- LONGITUD : The longitude of the location where the accident occurred. Values of 777.7777, 888.8888, 999.9999 will be considered null. NOTE: Yes, it is spelled “LONGITUD" without the ‘E‘ on the end.
- HOUR : Hour of the day when the accident occurred.
- DAY : The day of the month when the accidents happened.
- MONTH : The month the accident happened.
- YEAR : The year the accident happened.
- DRUNK_DR : Number of drunk drivers involved in the accident.
- FATALS : The number of fatalities in the accident.

The Vehicle File

The vehicle table is extremely rich. There is an entry for every vehicle involved in a fatal car accident. Many additional analyses could be performed on the data here, but we are interested in the following columns:

- ST_CASE : The unique case ID. This will be used primarily for merging information from the other tables.
- VEH_NO : The unique vehicle ID identifying each car involved in a given accident. This will also be used in merging information from the other tables.
- SPEEDREL : Whether or not speed was a factor in the car accident. In the tables from 2010, 2011, and 2012, the value 0 means No, and the value 1 means Yes. In the tables from 2013 and 2014, they begin classifying how speed was a factor. The value 0 still means No, and for our purposes, values from 2-5 mean Yes. Values of 8 or 9 mean Unknown.

The Person File

The person table is also extremely rich. Each entry contains information for all persons involved in the accident. For our analyses, we are interested in the following columns:

- ST_CASE : The unique case ID. This will be used primarily for merging information from the other tables.
- VEH_NO : The unique vehicle ID identifying each car involved in a given accident. This will also be used in merging information from the other tables.
- PER_TYP : The role of this person in the accident. Though the labels in this column vary greatly, we will only be interested in entries equal to 1, signifying the driver.
- DRINKING : Whether or not alcohol was involved for this person. The value 0 means No, 1 means Yes, and 8,9 mean Unknown.

The first task is to prepare the DataFrames will we use in our analyses, the `accidents` DataFrame and the `drivers` DataFrame. In the provided file `fars_data.zip`, you will find pickle files for accidents, vehicles, and persons corresponding to each fatal accident. These files are divided by year, but we want to combine the data from all these years. These pickle files can be loaded using `pd.read_pickle()`.

Problem 1. For the `accidents` DataFrame, we will need the columns:

```
ST_CASE, STATE, LATITUDE, LONGITUD, FATALS, HOUR, DAY, MONTH,
YEAR, DRUNK_DR, SPEEDING.
```

The SPEEDING column is derived from the SPEEDREL column of the vehicles file. The SPEEDREL column describes which vehicles specifically were speeding. We want the SPEEDING column of this table to be 0 if no cars were speeding and 1 if any of the cars involved were speeding. Also, remove all rows containing latitudes or longitudes that we are considering null due to the criteria described above.

The resulting DataFrame should be 149698 rows x 7 columns. As another indicator that you have done everything correctly, you should have a total of 44223 speed related accidents described in the SPEEDING column.

Lastly, the STATE column is all based on IDs. The `id_to_state.pickle` file contains a pickle file with a Python dictionary mapping IDs to states. Replace all the integer IDs in this column with the corresponding state abbreviation according to the mapping contained in this dictionary. To load this pickle file, execute the following code:

```
import pickle
with open("id_to_state.pickle") as file:
    id_to_state = pickle.load(file)
```

So in the end your DataFrame should look like this:

ST_CASE	STATE	LATITUDE	LONGITUD	HOUR	DAY
10001	AL	32.641064	-85.354692	4	15
10002	AL	31.430447	-86.956694	6	11
10003	AL	30.691631	-88.085778	15	14
10004	AL	33.868700	-86.291164	1	21
10005	AL	33.309742	-86.787222	6	4

MONTH	YEAR	DRUNK_DR	SPEEDING	FATALS
1	2010	1	0	1
1	2010	0	0	1
1	2010	0	1	1
1	2010	0	0	1
1	2010	0	0	1

Note

HINT: Though tedious, it will be easiest to do this whole process correctly if you do all the needed cleaning for each individual year, then concatenate all the years together into your final DataFrames. This is due to the fact that the IDs in the ST_CASE column are not year specific.

Problem 2. The map we will be using does not does not recognize latitude and longitude coordinates, but rather coordinates in meters. The FARS data set represents the location of the accidents in terms of latitude and longitude, so we will need to convert the coordinate system. This is done with the following code:

```
from_proj = Proj(init="epsg:4326")
to_proj = Proj(init="epsg:3857")

def convert(longitudes, latitudes):
    """Converts latlon coordinates to meters.

    Inputs:
        longitudes (array-like) : array of longitudes
        latitudes (array-like) : array of latitudes

    Example:
        x,y = convert(accidents.LONGITUD, accidents.LATITUDE)
    """

    x_vals = []
    y_vals = []
    for lon, lat in zip(longitudes, latitudes):
        x, y = transform(from_proj, to_proj, lon, lat)
        x_vals.append(x)
```

```
        y_vals.append(y)
    return x_vals, y_vals

accidents["x"], accidents["y"] = convert(accidents.LONGITUD, accidents.↩
    LATITUDE)
```

Problem 3. For the `drivers` DataFrame, we will need the columns:

```
ST_CASE, VEH_NO, PER_TYP, AGE, DRINKING, SPEEDREL, YEAR.
```

To obtain the information needed for this DataFrame, you will need merge the vehicles file and person file. You will want to merge on ST_CASE and VEH_NO.

The resulting DataFrame should be 341436 rows x 7 columns. It is easiest to add the YEAR column manually. Additionally, we will only be interested in the entries where PER_TYP is 1. This brings the final shape of the DataFrame to 223490 rows x 6 columns (since we can now eliminate PER_TYP).

Your DataFrame should look like this:

ST_CASE	VEH_NO	AGE	DRINKING	SPEEDREL	YEAR
10001	1	51	9	0	2010
10002	1	44	0	0	2010
10003	1	27	9	1	2010
10003	2	45	0	0	2010
10003	3	28	0	0	2010

Now with the necessary data cleaned, we can dive into the Bokeh package itself.

Basic Plotting

The general framework of Bokeh is built on Figures, Glyphs, and Charts. Figures are analogous to Figure objects in matplotlib. Glyphs consist of any lines, circles, squares, or patches that we may want to add to the Figure. Charts are visualizations such as bar charts, histograms, box plots, etc.

There are a few different ways to view your Bokeh plots, but we will address only two of them. These are `output_file()` and `output_notebook()`. If you choose to use `output_file`, your Bokeh plot will be saved to an HTML file that can be viewed in a web browser. This function accepts a string with the name of the desired filename. If you use `output_notebook()`, your Bokeh plots will appear in your Jupyter Notebook.

NOTE

There may be times that your plots may not be behaving as expected. If you are sure there is not a problem with your code, try restarting the Bokeh server. Assuming you are working in a Jupyter Notebook, this is done by first restarting the kernel, then reloading the webpage that hosts your Jupyter Notebook. Additionally, there is a currently a memory leak when using `output_notebook()`. After showing your plot several times in the Jupyter NOtebook, your notebook may crash because of losing memory. Again, the solution is restarting the kernel and refreshing the webpage.

As has been mentioned already, this lab is not meant to be exhaustive, but rather is meant to expose you to the basics. For our project, we will be using Circle glyphs, Square glyphs, Patch glyphs, and Bar charts. Here are some very basic examples of how to use each of these.

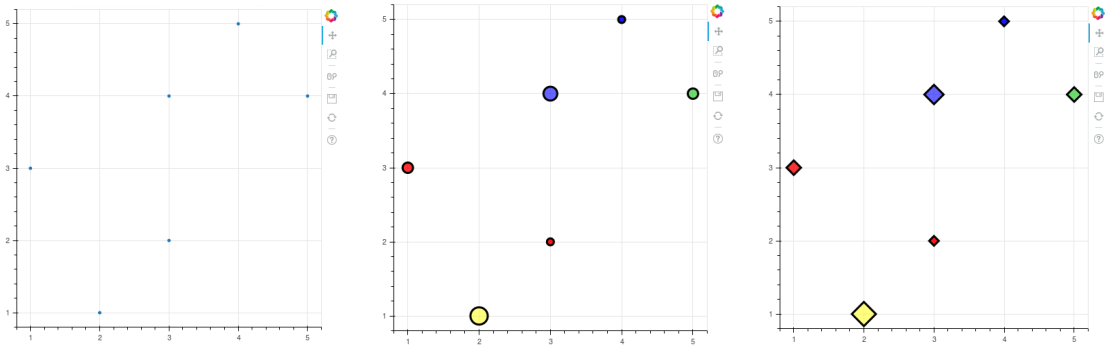
Marker Glyphs (Circles and Squares)

We use Circle glpyhs when we want to plot a collection of circles (such as in a scatter plot.) The following code example demonstrates the basic syntax of how this is done.

```
from bokeh.plotting import figure, output_file, show

output_file("my_plot.html")

fig = figure(plot_width=500, plot_height=500)

fig.circle(x=[1,2,3,3,4,5], y=[3,1,2,4,5,4])

show(fig)
```

The appearance of marker glyphs is high customizable. There are more than 20 keyword arguments that allow you to tweak the appearance of your markers. Here are some of the arguments that are most commonly used.

- size : size of marker measured in pixels on screen. So these will appear the same size despite the level of zoom.
- radius : size of marker measured by radius. These markers will scale with the level of zoom.
- fill_color : string of the hex value or the name of the color. Valid color names are all colors that have names in HTML.
- fill_alpha : value between 0 and 1 indicating alpha value. 0 indicates invisible, 1 indicates opaque.
- line_color : string of the hex value or the name of the color of the border. Valid color names are all colors that have names in HTML.
- line_alpha : value between 0 and 1 indicating alpha value of the border. 0 indicates invisible, 1 indicates opaque.
- line_width : thickness of the border.



(a) Default circles (b) Customized circles (c) Customized Squares

Additionally, you can pass lists of items to these keyword arguments. The attributes you specific in these list will be consistent across the indices of these lists.

```
fig = figure(plot_width=500, plot_height=500)

cir = fig.circle(x=[1,2,3,3,4,5], y=[3,1,2,4,5,4], size=[15,25,10,20,10,15],
            fill_color=["red", "yellow", "red", "blue", "blue", "limegreen"],
            fill_alpha=[.8,.5,.9,.6,.9,.7], line_color="black", line_width=3)

show(fig)
```

The syntax for squares is identical. In addition to the keyword arguments listed above, the `angle` keyword argument is also useful for squares.

```
fig = figure(plot_width=500, plot_height=500)

sq = fig.square(x=[1,2,3,3,4,5], y=[3,1,2,4,5,4], size=[15,25,10,20,10,15],
            fill_color=["red", "yellow", "red", "blue", "blue", "limegreen"],
            fill_alpha=[.8,.5,.9,.6,.9,.7], line_color="black", line_width=3,
            angle=np.pi/4)

show(fig)
```

Patch Glyphs

Patch glyphs are polygon shapes defined by a series of points locating the corners of the object. These shapes can be very complex. For our project, each state is a Patch glyph. The syntax for creating Patch glyphs is very similar to the syntax for creating markers. Most of the keyword arguments are the same as well.

```
# first iteration of the Sierpinski triangle.
fig = figure(plot_width=500, plot_height=500)

pats = fig.patches(xs=[[1, 3, 2], [3, 5, 4], [2, 4, 3]],
            ys=[[1, 1, 3], [1, 1, 3], [3, 3, 5]],
```

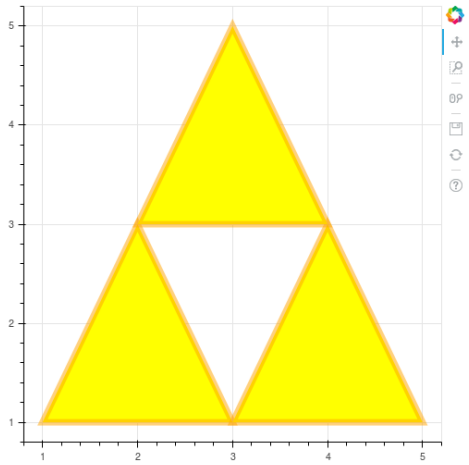


Figure 10.2: Collection of patch glyphs

```
            fill_color="yellow", line_color="orange",
            line_alpha=.5, line_width=7)

show(fig)
```

ColumnDataSource Object

In the plotting examples we have addressed up to this point, we have expressed the x and y values explicitly. As mentioned throughout this lab, one of the key features of Bokeh is to be to interact with your plots. However, if you have a large dataset, the performance of the interactions can be severely hampered.

Bokeh has a solution for this problem. It comes in the form of the `ColumnDataSource` object. We load our data into this object, connect our glyph to this object, then if necessary, Bokeh will automatically downsample the data to maintain acceptable performance. Once a glyph is linked to this source, you can also change the values in the source to update the positions and attributes of your glyph. This is an essential component to many of the key features of Bokeh.

ColumnDataSource objects accept a dictionary or a pandas DataFrame as the source of data. We reference the different data in this source by the associated key (in the case of a dictionary) or column name (in the case of a DataFrame).

```
# Replicate the circle marker plot using a DataFrame and ColumnDataSource.
from bokeh.models import ColumnDataSource

fig = figure(plot_width=500, plot_height=500)

df = pd.DataFrame({"x_vals":[1,2,3,3,4,5],
                   "y_vals":[3,1,2,4,5,4],
                   "size":[15,25,10,20,10,15],
                   "fill_color":["red", "yellow", "red", "blue", "blue", "↩
                       limegreen"],
                   "fill_alpha":[.8,.5,.9,.6,.9,.7]})
```

```
cir_source = ColumnDataSource(df)

cir = fig.circle(x="x_vals", y="y_vals", source=cir_source,
            size="size", fill_color="fill_color",
            fill_alpha="fill_alpha", line_color="black", line_width=3)

show(fig)


# Replicate the patches plot using a dictionary and ColumnDataSource.
fig = figure(plot_width=500, plot+height=500)

pat_data = dict("x_vals":[[1, 3, 2], [3, 5, 4], [2, 4, 3]],
                "y_vals":[[1, 1, 3], [1, 1, 3], [3, 3, 5]])

pat_source = ColumnDataSource(data=pat_data)

pats = fig.patches(xs="x_vals", ys="y_vals",
            fill_color="yellow", line_color="orange",
            line_alpha=.5, line_width=7)

show(fig)
```

Problem 4. We will start by adding a map to a Bokeh Figure. Here is the code you will need. This code will serve as the building block for the rest of this lab.

```
from bokeh.plotting import Figure
from bokeh.models import WMTSTileSource

fig = Figure(plot_width=1100, plot_height=650,
            x_range=(-13000000, -7000000), y_range=(2750000, 6250000),
            tools=["wheel_zoom", "pan"], active_scroll="wheel_zoom")
fig.axis.visible = False

STAMEN_TONER_BACKGROUND = WMTSTileSource(
    url='http://tile.stamen.com/toner-background/{Z}/{X}/{Y}.png',
    attribution=(
        'Map tiles by <a href="http://stamen.com">Stamen Design</a>, '
        'under <a href="http://creativecommons.org/licenses/by/3.0">CC BY ↩
           3.0</a>.'
        'Data by <a href="http://openstreetmap.org">OpenStreetMap</a>, '
        'under <a href="http://www.openstreetmap.org/copyright">ODbL</a>'
    )
)

fig.add_tile(STAMEN_TONER_BACKGROUND)
```

Problem 5. Using Patch glyphs, draw all the borders for all the states. The border information you need is found in the pickle file `borders.pickle`. This pickle file contains a nested Python dictionary where the key is the two letter abbreviation for a given state and the value is a dictionary containing a list of latitudes and a list of longitudes.

We need a list of lists of x-values and y-values to pass to the `fig.patches()` function. You will need to convert these coordinates the same way you converted the coordinates in Problem 2

To get a list of lists of latitudes and longitudes, you may use the following code:

```
state_xs = [us_states[code]["lons"] for code in us_states]
state_ys = [us_states[code]["lats"] for code in us_states]
```

Draw the state borders using the coordinates defined in `state_xs` and `state_ys` and using a `ColumnDataSource` object.

Problem 6. Your `accidents` DataFrame should contain the converted longitudes and latitudes for all the fatal accidents. In the same figure from Problem 5, plot a circle for each fatal car accidents. For this problem, you should use your `accidents` DataFrame in conjuction with a ColumnDataSource. For an added level of detail, color the markers depending on the type of accident: drunk, speeding, other. The best way to do this would be to have 3 different `ColumnDataSource` objects.

Speeding up Interactions with WebGL

As you pan around your figure from Problem 6, it may take a few seconds to load after each movement. This may not be that surprising considering you have just added approximately 150,000 circles to your figure. The performence can be improved using WebGL. WebGL stands for Web Graphics Library. It takes advantage of a compmuter's GPU when plotting a large number of points.

Problem 7. To take advantage of WebGL, add `webgl=True` as a keyword argument to your figure. You should notice a significant improvement in the response time while panning and zooming.

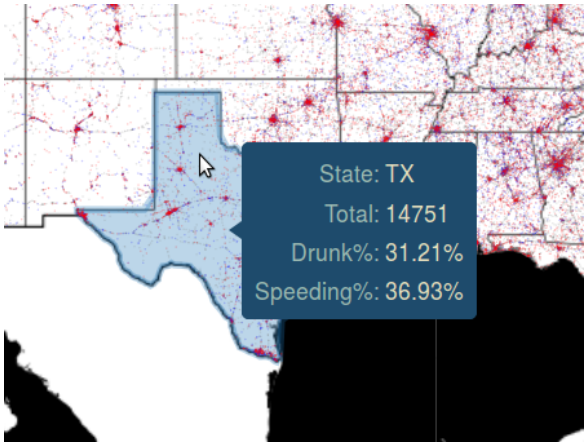


Figure 10.3: Example of state hover tooltip.

> **Note**
>
> There are times that adding WebGL support to your figure will cause it to behave strangely. WebGL support is still a fairly new feature of Bokeh. Remember that Bokeh is still in development, so hopefully things like this will be refined in time.

Adding Tooltips

It would be useful to display a bit more information on our map. We will accomplish this with tooltips. When we hover over each state, we will display a few pieces of key data that may be interesting to the viewer. To be able to show more information, we will need to prepare a few more pieces of data.

Problem 8. In Problem 5, you created two lists of lists, `state_xs` and `state_ys`. When hovering over a Patch glyph, Bokeh keeps track which index corresponds with that Patch. For our tooltips, we will need a few more lists. We have to take extra special care that the indices of all these lists are consistent.

Using list comprehension again, create a list of state abbreviations, a list of the total number of accidents by state, a list of the percentage of speed-related accidents by state, and a list of the percentage of drunk driving accidents by state.

There is more than one correct way to do this, but whatever method you choose to use, take extra care to make sure that the indices correspond to the same state across all these lists.

HoverTool and Tooltips

Adding tooltips to Patch glyphs is fairly straightforward. The process is best explained by first presenting an example.

```
import numpy as np
from bokeh.plotting import figure, output_notebook, show
from bokeh.models import HoverTool, ColumnDataSource

x_vals = [[0,1,0], [0,1,1], [1,2,1], [1,2,2]]
y_vals = [[0,0,1], [1,1,0], [0,0,1], [1,1,0]]

x_coords = [0,0,1,1,2,2]
y_coords = [0,1,0,1,0,1]

fig = figure(plot_width=500, plot_height=300)
col = ["Blue", "Red", "Yellow", "Green"]

patch_source = ColumnDataSource(
    data=dict(
        col=col,
        x_vals=x_vals,
        y_vals=y_vals
    )
)

circle_source = ColumnDataSource(
    data=dict(
        x_coords=x_coords,
        y_coords=y_coords
    )
)

triangles = fig.patches("x_vals", "y_vals", color="col", source=patch_source,
           line_color='black', line_width=3, fill_alpha=.5, line_alpha=0
           hover_color="col", hover_alpha=.8, hover_line_color='black')
circles = fig.circle("x_coords", "y_coords", fill_color='black', source=↩
    circle_source,
           fill_alpha=.5, hover_color="black", hover_alpha=1, line_alpha=0, ↩
               size=18)

fig.add_tools(HoverTool(renderers=[triangles], tooltips=[("Color", " @col")]))
fig.add_tools(HoverTool(renderers=[circles], tooltips=[("Point", " (@x_coords, ↩
    @y_coords)")]))

show(fig)
```

Now let's piece apart this example. Notice first that we have a `ColumnDataSource` object for the patch coordinates and another `ColumnDataSource` for the circles. In general, it is a good idea to separate `ColumnDataSource` objects like this, but more specifically, we need to do this because we want to have two different hover behaviors.

Next, notice that there are some keyword arguments included in these glyphs we have not discussed yet. The keyword arguments `hover_color`, `hover_alpha`, and `hover_line_color`. When hovering over one of these glpyhs, these arguments overwrite `fill_color`, `fill_alpha`, and `line_color` , respectivley.

Finally, when creating the `HoverTool` object, you will most commonly use the `renderers` and `tooltips` keyword arguments. The `renderers` argument accepts a list of glyphs. At times, there is unpredicted behavior if you include more than one glpyh in this list. The `tooltips` arguments functions just like the `tooltips` argument we discussed in the section on bar charts.

Problem 9. On your plot of the United States, we can provide much more information to this plot using tooltips. Using the lists you prepared in Problem 8, add tooltips for the Patch objects in this plot. Also adjust the appearence of the Patch objects under the mouse so it is clear which state is being selected.

Your tooltips should look similar to Figure 10.3.

Adding More Complicated Interactions Using Widgets

> NOTE
>
> At the beginning of this lab, we mentioned that Bokeh is still in development so some explanations will need to be updated. There is some material in this section that is still under active development. In theory, the code presented here will still work, but it is quite possible that there will be a better way to do these tasks in the future.

One of the mottos the Bokeh developers have is, *We write the Javascript so you don't have to.* If you happen to have experience with Javascript, you can do some pretty amazing things with Bokeh interactions. In this section, we address some of the interactions that are possible without using Javascript.

Select

The Select widget is ideal for changing attributes of the figure where you want to allow only a few different options. The code for creating Select widgets is very straightforward.

```
from bokeh.io import output_file, show
from bokeh.models.widgets import Select

output_file("select.html")

select = Select(title="Option:", value="one",
                options=["one", "two", "three", "four"])

show(select)
```

For our project, we want to be able to give the user the ability to gain information from the first map without needing to hover over each individual state. Since we are interested in drunk driving accidents and speeding accidents, it makes sense to give the user the option to color the states according to the percentage of these types of accidents.

To be able to accomplish this, we will need to have some way of knowing if the value in the Select widget has changed, and if it has, what we should do with the new value.

Most Bokeh widgets have a `on_change` method. This method is called whenever a certain parameter (usually `"value"`) is changed. Here is a simple example.

```
import pandas as pd
import numpy as np
from bokeh.io import curdoc
from bokeh.plotting import Figure
from bokeh.models import ColumnDataSource, Select
from bokeh.layouts import column

COUNT = 10
df = pd.DataFrame({"x":np.random.rand(COUNT),
                   "y":np.random.rand(COUNT),
                   "color":"white"})
source = ColumnDataSource(df)

fig = Figure()
fig.circle(source=source, x="x", y="y", fill_color="color",
           line_color="black", size=40)

select = Select(title="Option:", value="white",
                options=["white", "red", "blue", "yellow"])

def update_color(attrname, old, new):
    source.data["color"] = [select.value]*COUNT

select.on_change('value', update_color)

curdoc().add_root(column(fig, select))
```

There are a few things to point out with this example. Note that we have a function called `update_color`. This function is called whenever the `"value"` of the Select widget is changed. The function arguments `attrname, old, new` are used by Bokeh, but you don't need to worry about them at all in this lab. Because we have our Circle glyph tied to a `ColumnDataSource`, any change to this source will affect the Circle glpyh.

Second, note that we have not specified an `output_file`. Instead, we use `curdoc()`, which stands for "current document". We add the layouts we want to the current document through the `add_root()` method. In our case, we stack all the elements of our document using the `column()` function. You can learn more about possible layouts here: `http://bokeh.pydata.org/en/latest/docs/user_guide/layout.html`

Using the `on_change` function requires a Bokeh Server to be running. You can view the code above by executing

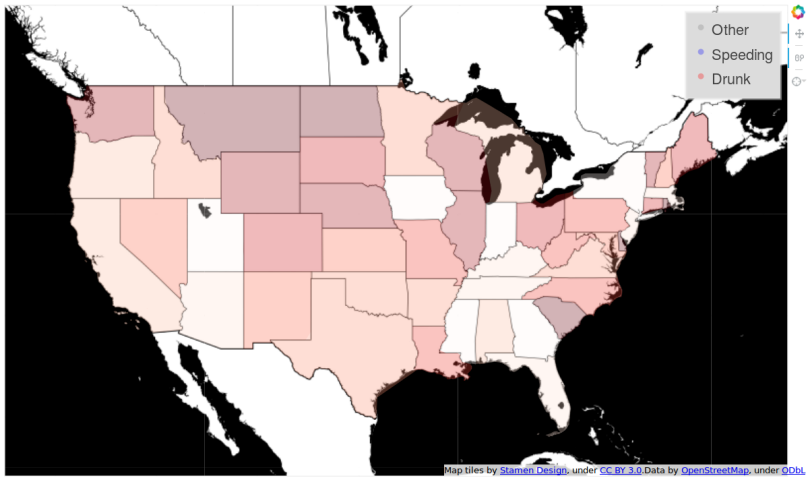


Figure 10.4: Example of coloring states based on percentage of drunk driving fatalities.

```
$ bokeh serve <FILENAME>.py
```

in your terminal, then going to `localhost:5006/<FILENAME>` in your web browser.

Problem 10. In this problem, we will add a Select widget to our map. This Select widget will specify the color map we wish to use for the states.

In your Select widget, have options for `"None"`, `"Drunk %"`, and `"Speeding %"`. To assign the states the appropriate color, you may use the following code, or you may adjust the functionality as you like.

```
# change this first line if you want a different colormap
from bokeh.palettes import Reds9

COLORS.reverse()
no_colors = ['#FFFFFF']*len(state_names)
drunk_colors = [COLORS[i] for i in pd.qcut(state_percent_dr, len(COLORS)).↩
    codes]
speeding_colors = [COLORS[i] for i in pd.qcut(state_percent_sp, len(COLORS↩
    )).codes]
```

This code assumes you have lists named `state_names`, `state_percent_dr`, and `state_percent_sp` from Problem 8.

Changing the values of the select box should result in something similar to Figure 10.4.

11 Web Technologies 1: Internet Protocols

Lab Objective: *The internet has strict protocols for managing communications between machines. In this lab, we introduce the basics of TCP, IP, and HTTP, and create a server and client program. We then apply this understanding to navigation of online infrastructure and to data collection.*

The internet is comparable to a network of roads connecting the buildings of a city. Each building represents a computer and the roads are the physical wires or wireless pathways that allow for intercommunication. In order to properly navigate the road, you must use the correct kind of vehicle and follow the established instructions for travel. If these requirements are not met, you are not allowed to navigate the road to another building. There are also vehicles of different capabilities and sizes which can retrieve items from particular buildings. In a similar fashion, the internet has specific rules and procedures, called protocols, which allow for standardized communication within and between computers. By understanding these protocols, we can more easily navigate the world web to collect data, create web dependent programs, and interact with other network connected computers.

The most common communication protocols in computer networks are contained in the Internet Protocol Suite. Among these are TCP (Transmission Control Protocol) and IP (Internet Protocol), which are two of the oldest and most important protocols. In fact, they are so important that the entire suite is sometimes referred to as TCP/IP. However, there are many other protocols in the suite, all of which are divided into four abstraction layers:

1. **Application**: Software that utilizes transport protocols to move information between computers. This layer includes protocols important for email, file transfers, and browsing the web.

2. **Transport**: Protocols that define basic high level communication between two computers. The two most common protocols in this layer are TCP and UDP. TCP is by far the most widely used due to its reliability. UDP, however, trades reliability for low latency.

3. **Internet**: Protocols that handle routing and movement of data on a network.

4. **Link**: Protocols that deal with local networking hardware such as routers and switches.

For this lab, we will focus on understanding and utilizing TCP/IP and HTTP, which are the most common Transport and Application protocols, respectively.

TCP

TCP is specifically used to establish a connection between computers, exchange bits of information called *packets*, and then close their connection. Built for host computers on an IP network, TCP allows for a very reliable, ordered, and error-checked stream of data bytes (eight binary bits).

Specifically, TCP creates network *sockets* that are used to send and receive data packets. While we would normally think of sending data between two different machines, two sockets on the same machine can communicate via TCP as well. Using the Python `socket` module, we will demonstrate how to create a network socket (a *server* to listen for incoming data, and how to create a second socket (a *client*) to send data.

Creating a Server

A *server* is a program that provides functionality to *client* programs. Oftentimes, these server programs run on dedicated computer hardware that we also refer to as servers. These servers are fundamental to the modern networks we work on and provide services such as file sharing, authentication, webpage information, databases, etc. To create a server, we first create a new socket object. The socket object will be able to listen for and accept incoming connections from other sockets.

The two input arguments specify the socket type. Further description of socket types can be found in the python documentation.

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

We then define an address and a port for the socket to listen on. A port is analogous to a mailbox on the computer. There are 65535 available ports. Of those, about 250 are commonly used. Certain ports are have pre-defined uses. For example:

- 0 to 1023: Special reserved ports
- 80, 443: Commonly used for web traffic
- 25, 110, 143, 465: Commonly used for email servers

We also specify an address for the host, which is analogous to the "mailing address" of the machine on which the server is running. The address may be set to the computer's IP address, to `"localhost"`, or to an empty string. We bind the socket to the port and address, then call `listen()` to tell it to listen for connections.

```
address = '0.0.0.0' # Default address that specifies the local machine and ↩
    allows connection on all interfaces
s.bind((address, 33498)) # Bind the socket to a port 33498, which was ↩
    arbitrarily chosen
s.listen(1) # Tell the socket to listen for incoming connections
```

Next we tell the socket what do with incoming connections. Once a connection is made, the `accept()` method returns the connection, which is itself a socket object. The connection object receives data (as a string) in blocks, so we specify a block size in bits.

The connection object can also send back data. In the code below, the connection simply echoes back whatever data it receives. After all the data has been received, we close the connection.

```
size = 2048 # Block size of 20 bytes
conn, addr = s.accept() # conn is our new socket object for receiving/sending ↩
    data
print "Accepting connection from:", addr
while True:
    data = conn.recv(size) # Read 20 bytes from the incoming connection
    if not data: # Terminate the connection if data stops arriving (no more ↩
        blocks to receive)
        break
    conn.send(data) # Send the data back to the client
conn.close()
```

We can also close the server by using the KeyBoardInterrupt (`ctrl+c`).

ACHTUNG!

When running the code above, you will see the program hang on the line,

```
conn, addr = s.accept()
```

As mentioned above, the `accept()` method does not return until a connection is made. Therefore for the code above to execute in its entirety, a client needs to connect to the server. Creating a client is addressed in the next section.

Creating a Client

A *client* is a program that contacts a server in order to receive data or functionality. We use many client programs which include web browsers, mail programs, online video games, etc. We will create a new client socket to connect to our server. We follow a similar process as we did for the server socket:

```
import socket
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

We specify the address of the server, and the port (this needs to be the same port on which the server is listening). We then connect to the server.

```
ip = '0.0.0.0'
port = 33498
client.connect((ip, port))
```

Once connected, the client can send and receive data. Unlike the server, the client sends and reads the data itself instead of creating a new connection socket. When we are done with the client, we close it.

```
size = 2048 # Block size of 20 bytes
msg = "Trololololo, lolololololo, ahahahaha."
```

```
client.send(msg)

print "Waiting for the server to echo back the data..."
data = client.recv(size)
print "Server echoed back:", data

client.close()
```

To see the client and server communicate, open a terminal and run the server. Then run the client in a separate terminal.

Command	Description
`bind((address, port))`	Binds to a port and an address.
`listen()`	Starts listening for requests.
`accept()`	Accepts a connection from a client, and returns a new socket object and a connection address.
`recv(size)`	Reads and returns a block of incoming data.
`send(data)`	Sends data to the client.
`close()`	Closes the socket.
`gethostname()`	Returns the host name of the machine.
`getsockname()`	Returns the socket's own address.

Table 11.1: Table of Socket Commands

Problem 1. Write a file called `simple_server.py`. When run, this file should create a server socket, accept a connection and then read incoming data. The server should append the current time onto each data block, then send it back to the client.
(Hint: use `time.strftime('%H:%M:%S')` to format the current time nicely.)

Also write a file called `simple_client.py`. In this file, create a client socket and connect to the server created in `simple_server.py`. The client should send a message to the server and print the server's response.

Problem 2. Write a file called `rps_server.py`, which plays rock-paper-scissors with a client. The server should accept a connection, and while the connection is open, cycle through the following loop:

- Receive a move (`"rock"`, `"paper"`, or `"scissors"`).
- Generate a random move of its own. Print both moves.
- Determine who won the round.
- Send `"you win"`, `"you lose"`, or `"draw"` back to the client, depending on the outcome of the round. If the move is invalid, send back `"invalid move"`.
- If the client won, break the loop and close the connection.

Also write a file called `rps_client.py`. The client should connect to the server and then enter a while loop. In the loop, the client sends the server a move and prints the server's response. The client should break the loop and close once it receives a `"you win"` back from the server.

Although these examples are simple, we use a similar pattern for every transfer of data over TCP. For simple connections, the amount of work the programmer has to do can be minimal. However, requesting a complicated webpage would require us to manage possibly hundreds of connections. Naturally, we would want to use a higher level protocol that takes care of the smaller details for us.

HTTP

HTTP stands for Hypertext Transfer Protocol, which is an application layer networking protocol. It is a higher level protocol than TCP and takes care of many of the small details of TCP for us. It also relies on underlying TCP protocol to provide network capabilities. The protocol is centered around a request and response paradigm. A client makes a request to a server and the server replies with response. There are several methods, or requests, defined for HTTP servers. The three most common of which are GET, POST, and PUT. GET requests are typically used to request information from a server. POST requests are sent to the server with the intent of modifying the state of the server. PUT requests are used to add new pieces of data to the server.

Every HTTP request consists of two parts: a header and a body. The headers contain important information about the request including: the type of request, encoding, and a timestamp. We can add custom headers to any request to provide additional information. The body of the request contains the requested data and may or may not be empty.

We can setup an HTTP connection in Python as demonstrated below. We will encourage you to use the `requests` library instead of the modules in the standard library. However, the code below is illustrative of the steps in making an HTTP connection using `httplib`.

```
import httplib
conn = httplib.HTTPConnection("www.example.net")
conn.request("GET", "/")
resp = conn.getresponse() # Server response message
if resp.status == 200:
    headers = resp.getheaders()
    data = resp.read()
conn.close() # After collecting the information we need, we close the ↩
    connection
print headers
print data      # Long string full of HTML code from the webpage
```

The above codes starts by creating a connection to specific host. We then make a request, which in this case was a GET request. The host we are connected to then responds and we retrieve the response. In order to know if our request was successfully processed, we need to check the response message from the server. A status code of 200 means that everything went alright. We can now read the data of the response and then explicitly close the connection.

This exchange is greatly simplified by the `requests` library:

```
import requests
```

```
r = requests.get("http://www.example.net")
r.close()
print r.headers
print r.content
```

We will now examine an example of a GET request with a list of parameters in the form of a Python dictionary. We will use a web service called HTTPBin which is very helpful in developing applications that make HTTP requests.

```
>>> data = {'key1': 0, 'key2': 1}
>>> r = requests.get("http://httpbin.org/get", params=data)
>>> print r.content # Our parameters now show up in the args() of the get ↩
    request
```

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

Problem 3. The file `nameserver.py` is an example of a simple HTTP server using the Python module `BaseHTTPServer`. It allows clients to send the last name of a famous computer scientist with GET and then returns the corresponding first name. For example, running the following code results in the message `"Grace"` from the server. The *?* signifies the start of a *query* in the URL.

```
requests.get("http://localhost:8000?lastname=Hopper").text
```

Expand the functionality of the server to do the following:

- Obtain a list of everybody whose last name starts with a given string. For example, a query of `lastname=B` would return all the names whose last name starts with the letter `"B"`.
- Similarly, obtain a list of everybody in the list of names with `lastname=AllNames`.
- Then add a method `do_PUT()` so that clients can add a person to the dictionary (or modify an existing entry) using PUT with a query of `firstname` and `lastname`. Multiple query fields separated using the `"&"` character.

The format of the list of names returns from the server should be:

```
LASTNAME, FIRSTNAME
LASTNAME, FIRSTNAME
   ...       ...
```

Though simple, this HTTP server was the original idea for Instant Messaging. The instant messaging client on your computer sends your name, IP address, and port number to the web server.

This server then advises your contacts that you are online. If they wish to message you, all communication then happens between your port and IP address and the port and IP address of the other user without passing through the main name server.

Problem 4. We will now practice chatting through HTTP. To do so, create a new HTTP server in a file called `chatserver.py` using the same method as Problem 3. This new server should store a dictionary with keys of names and values as lists of messages. The GET method should return in this format:

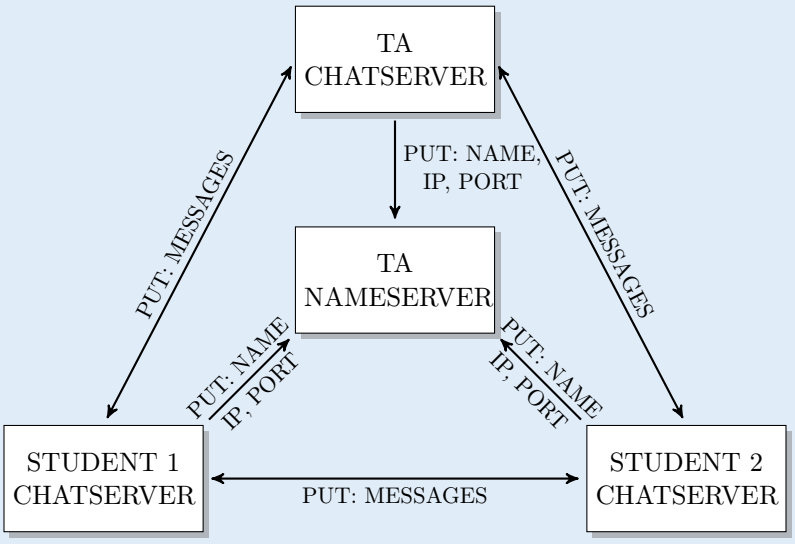
```
Name1:
    Message 1
    Message 2
Name2:
    Message 1
    ...
...
```

The PUT method should accept a `name` and a `message` and store them in your dictionary.

Once you have successfully created your server, run it through a terminal at your localhost IP address. You may then PUT your `name` and `ipaddressport` on the HTTP server provided by your instructor. To find your own IP address, run in a separate terminal, `ifconfig` (for Mac or Linux) or `ipconfig` on Windows command prompt. You may use GET requests of `AllNames` to the nameserver to retrieve the names of other active users. To send them a message, use PUT requests to their respective IP address and port.

Sample requests are as follows:

```
# Request names and IP addresses from nameserver.
r = requests.get("http://ipaddress:portnumber?name=AllNames")
# Send a message to another persons chat server.
r = requests.put("http://ipaddress:portnumber?name=Thomas&message=Hello ↩
    World")
```



To receive credit for this problem, please do the following:

1. PUT your name, IP address, and port number on the provided nameserver.
2. Send a message with your name to the *TA* user IP address and port.
3. Accomplish any additional tasks specified by your instructor.

When you have finished chatting, update your IP address to an empty string in the name server.

Achtung!

This problem requires a level of remote network access. If the network doesn't allow you to connect, use SSH to remote into a network location where this can be done.

12 Web Technologies 2: Data Serialization

Lab Objective: *Understand how serialization is used in web technologies. Practice using serialization to pack, transport, unpack, and navigate data structures in order to more easily utilize web based data sources.*

In order to more easily store and exchange data structures, standardized metalanguages have been developed to *serialize* data. Serialization is the process of packaging data with special properties in a form that can be easily unpacked and reconstructed as an identical copy on any computer. For example, if you wanted to share a k-d tree filled with data, you can easily send and replicate it on a colleague's computer by using serialization without having to send raw data and run the k-d sorting algorithm again. This can be used to transport exact data structures, or just to send data in an organized fashion, even between different programming languages. There are many different kinds of serialization methods for different languages and with different purposes, however, we will focus on serialization with the XML and JSON languages. These are two of the most prevalent serialization languages used for web communication and web design, but are commonly used to transport all programming languages. Their main application in web communication is in the transportation of organized data.

JSON

JSON stands for *JavaScript Object Notation*. This serialization method stores information about the objects as a specially formatted string that is easy for both humans and machines to read and write. When JSON is deserialized, this special string is parsed and the objects are recreated. Despite its name, it is a completely language independent format. JSON is built on top of two types of data structures: a collection of key/value pairs and an ordered list of values. In Python, these data structures are more familiarly called dictionaries and lists respectively.

The following is a very simple example of the characteristics of a family expressed in JSON:

```
{
    "lastname": "Smith"
    "father": "John",
    "mother": "Mary",
    "children": [
        {
            "name": "Timmy",
```

```
            "age": 8
        },
        {
            "name": "Missy",
            "age": 5
        }
    ]
}
```

> **Note**
>
> You have likely been working very closely with JSON without even knowing it! Jupyter Notebooks are actually stored as JSON. To see this, open one of your `.ipynb` files in a basic text editor.

In general, the JSON libraries of various languages have a fairly standard interface. Though the Python standard library has modules for JSON, if performance is critical, there are additional modules for JSON that are written in C such as `ujson` and `simplejson`.

Serialization using JSON

Let's begin by serializing some common Python data structures.

```
>>> import json
>>> ex1 = [0, 1, 2, 3, 4]
>>> json.dumps(ex1)
'[0, 1, 2, 3, 4]'
>>> ex2 = {'a': 34, 'b': 483, 'c':"Hello JSON"}
>>> json.dumps(ex2)
'{"a": 34, "c": "Hello JSON", "b": 483}'
```

The JSON representation of a Python list and dictionary are very similar to their respective string representations. Each of these generated strings is called a JSON *message*. Since JSON is based on a dictionary-like structure, you can nest multiple messages by distributing them appropriately within curly braces.

```
>>> aJSONstring = """{"car": {
    "make": "Ford",
    "model": "Focus",
    "year": 2010,
    "color": [255, 30, 30]
        }
    }"""
>>> t = json.loads(aJSONstring)
>>> print t
{u'car': {u'color': [255, 30, 30], u'make': u'Ford', u'model': u'Focus', u'year↩
    ': 2010}}
```

```
>>> print t['car']['color']
[255, 30, 30]
```

To generate a JSON message, use `dump()`. This method accepts a Python object, generate the message, and writes it to a file. The method `dumps()` does the same, but returns the string instead of writing it to a file. To perform the inverse operation, use `load()` or `loads()` to read a file or string, respectively.

The built-in JSON encoder/decoder only has support for the basic Python data structures such as lists and dictionaries. Trying to serialize an object which is not recognized will result in an error. Below is an example trying to serialize a set.

```
>>> a = set('abcdefg')
>>> json.dumps(a)
---------------------------------------------------------------------------
TypeError: set(['a', 'c', 'b', 'e', 'd', 'g', 'f']) is not JSON serializable
```

The serialization fails because the JSON encoder doesn't know how it should represent the set. However, we can extend the JSON encoder by subclassing it and adding support for sets. Since JSON has support for sequences and maps, one easy way would be to express the set as a map with one key that tells us the data structure type, and the other containing the data in a string. Now, we can encode our set.

```
class CustomEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, set):
            return {'dtype': 'set',
                    'data': list(obj)}
        return json.JSONEncoder.default(self, obj)

>>> a = set('abcdefg')
>>> json.dumps(a, cls=CustomEncoder)
'{"dtype": "set", "data": ["a", "c", "b", "e", "d", "g", "f"]}'
```

Though this is helpful, decoding this string would give us all of our data in a list. To allow for this string to be decoded as a Python set, we must build a custom decoder. Notice that we don't need to subclass anything.

```
>>> accepted_dtypes = {'set': set}
>>> def CustomDecoder(item):
...     type = accepted_dtypes.get(item['dtype'], None)
...     if type is not None and 'data' in item:
...         return type(item['data'])
...     return item

>>> c = json.loads(s, object_hook=CustomDecoder)
{u'a', u'b', u'c', u'd', u'e', u'f', u'g'}
# The 'u' is a prefix that signifies that the string value is Unicode.
# You can test this with:
>>> print c[0]
```

```
a
```

Problem 1. Python has a module in the standard library that allows easy manipulation of times and dates. The functionality is built around a `datetime` object.

```
>>> import datetime
>>> now = datetime.datetime.now()
>>> print now
2016-10-04 11:52:33.513885

# We can also extract the individual time units
>>> now.year
2016
>>> now.minute
33
>>> now.microsecond
513885
```

However, `datetime` objects are not JSON serializable. Determine how best to serialize and deserialize a datetime object, then write a custom encoder and decoder. The datetime object you serialize should be equal to the datetime object you get after deserializing.

APIs and JSON

Many websites and web APIs (Application Program Interface) make extensive use of JSON. For example, almost any programs that utilize Twitter, Facebook, Google Maps, or YouTube communicate with their APIs using JSON. This means that any website that uses an embedded version of Google Maps will receive JSON strings with data to display Google Maps interface on a portion of the webpage. It also allows developers to receive information and update the embedded portions of their page without needing to reload the webpage file. An example of this is available at `https://developers.google.com/maps/documentation/javascript/examples/map-simple`. There are also web APIs which allow developers to retrieve the website data in JSON strings. A list of APIs for public data collection can be found at `http://catalog.data.gov/dataset?q=-aapi+api+OR++res_format%3Aapi#topic=developers_navigation`

Achtung!

Each website has a policy about data usage and automated retrieval that requires certain behavior. If data is scraped without complying with these requirements, there can be legal consequences.

Problem 2. JSON files are often used by APIs to respond to data requests. To demonstrate an example of this, we will do a brief data examination of water usage in Los Angeles, California. The City of Los Angeles publishes some water usage data for the public. The API *endpoint* at which this data is available is at the address `https://data.lacity.org/resource/v87k-wgde.json`.

We can use the `requests` library from the previous lab to `GET` the data from the page.

```
requests.get("https://data.lacity.org/resource/v87k-wgde.json").json()
```

This code with load the object from JSON format into a Python list. Once the list has been created, gather the water usage data from 2012 to 2013 and the longitude and latitude of each point.

Use Bokeh to plot these points on a map of Los Angeles (refer to Lab 10 for additional help on Bokeh).

To draw a map centered on Los Angeles, you may use the following code:

```
from bokeh.plotting import figure
from bokeh.models import WMTSTileSource

fig = figure(title="Los Angeles Water Usage 2012-2013", plot_width=600, ↩
    plot_height=600, tools=["wheel_zoom", "pan", "hover"],
             x_range=(-13209490, -13155375), y_range=(3992960, 4069860),
              webgl=True, active_scroll="wheel_zoom")
fig.axis.visible = False

STAMEN_TONER_BACKGROUND = WMTSTileSource(
    url='http://tile.stamen.com/toner-background/{Z}/{X}/{Y}.png',
    attribution=(
        'Map tiles by <a href="http://stamen.com">Stamen Design</a>, '
        'under <a href="http://creativecommons.org/licenses/by/3.0">CC BY ↩
            3.0</a>.'
        'Data by <a href="http://openstreetmap.org">OpenStreetMap</a>, '
        'under <a href="http://www.openstreetmap.org/copyright">ODbL</a>'
    )
)

background = fig.add_tile(STAMEN_TONER_BACKGROUND)
```

To convert the longitude and latitude locations to be compatible with the map, you may the following code:

```
from pyproj import Proj, transform

from_proj = Proj(init="epsg:4326")
to_proj = Proj(init="epsg:3857")
```

```
def convert(longitudes, latitudes):
    x_vals = []
    y_vals = []
    for lon, lat in zip(longitudes, latitudes):
        x, y = transform(from_proj, to_proj, lon, lat)
        x_vals.append(x)
        y_vals.append(y)
    return x_vals, y_vals
```

More information about this dataset is available at `http://catalog.data.gov/dataset/residential-water-usage-zip-code-on-top-cb2ac`.

XML

XML is another data interchange metalanguage. However, it is a markup language rather than a object notation language. This means that it utilizes tags to distinguish the formatting of data rather than just encoding them. So, broadly speaking, XML is somewhat more robust and versatile than JSON, but slightly less efficient. Due to this minor difference, it is utilized in different ways.

To understand XML, we need to further explore what tags are. A tag is a special command enclosed in angled brackets (< and >) that describes properties of the data enclosed. For example, we can represent a car's properties in the XML below. Notice that tags are used to both open and close a property.

```
<car>
    <make>Ford</make>
    <model>Focus</model>
    <year>2010</year>
    <color model='rgb'>255,30,30</color>
</car>
```

XML data can be read as a tree or as a stream.

Since XML is a hierarchical storage format, it is very easy to build a tree of the data. The advantage of a tree format is random access to any part of the document at any time. However, this requires all of the XML to be loaded into memory for construction of the tree.

Reading a document as a stream means reading each piece sequentially or only reading a small portion of the file at a time. Because memory usage is constant, there is no limit to size of XML document that we can process this way, however, we do not have the random access of a tree.

The following will explore two APIs that parse XML formatted files and strings.

DOM

The DOM (Document Object Model) API allows you to work with an XML document as a tree in a hierarchy of elements. In order to sort the tree, the XML tags are read from the file and distributed accordingly. The DOM tree of the car from the XML code above would have `<car>` at the root element. This root element would have four children, `<make>`, `<model>`, `<year>`, and `<color>`. After a DOM tree has been sorted, we can traverse it like any other tree structure or search it by tag. Python's XML module includes two version of DOM: `xml.dom` and `xml.minidom`. MiniDOM is a minimal, more simple implementation of the DOM API.

SAX

SAX, Simple API for XML, is a very fast and efficient way to read an XML file. The main advantage of this method is memory conservation. A SAX parser will read XML sequentially instead of all at once. As the SAX parser iterates through the file, it emits events at either the start or the end of tags. You can provide functions to handle these events.

ElementTree

ElementTree is Python's unification of DOM and SAX APIs into a single, high-level API for parsing and creating XML trees. ElementTree provides a SAX-like interface for reading XML files via its `iterparse()` method. This will have all the benefits of reading XML via SAX. In addition to stream processing the XML, it will build the DOM tree as it iterates through each line of the XML input. ElementTree provides a DOM-like interface for reading XML files via its `parse()` method. This will create the tag tree that DOM creates.

We will demonstrate ElementTree using the following XML file.

```
<?xml version="1.0"?>
<contacts>
    <person>
        <firstname>John</firstname>
        <lastname>Doe</lastname>
        <phone type="mobile">1234567890</phone>
        <phone type="home">5432229875</phone>
        <email type="home">doughboy@bakery.com</email>
        <address type="home">34 South Street, Jonesville</address>
        <groups>personal,work</groups>
    </person>
    <person>
        <firstname>Sally</firstname>
        <lastname>Sue</lastname>
        <phone type="mobile">8372289491</phone>
        <groups>personal</groups>
    </person>
    <person>
        <firstname>Thor</firstname>
        <lastname></lastname>
        <phone type="mobile"></phone>
        <email type="home"></email>
```

```
        <address type="home"></address>
        <groups>work</groups>
    </person>
</contacts>
```

contacts.xml

First, we will look at viewing an XML document as a tree similar to the DOM model described above.

```
import xml.etree.ElementTree as et

f = et.parse('contacts.xml')

# To manually traverse the tree:
# We iterate through the element directly
# Note: getchildren() is old and deprecated (not supported), so we instead use list() to gather children.
root = f.getroot()
children = list(root) # root has three children
person0 = children[0]
fields = list(person0) # The children elements of person0
```

We can search the entire tree for specific elements by:

```
# Searching for all tags equal to firstname
for n in root.iter('firstname'):
    print n.text
```

We can also filter with multiple tags by:

```
seek = {'firstname', 'lastname', 'phone'}
fi = (x for x in root.iter() if x.tag in seek)
for n in fi:
    print n.text
```

We can even modify the document tree in place.

```
# To remove Thor:
for n in root.findall("person"):
    if n.find("firstname").text == 'Thor':
        root.remove(n)

# Verify that Thor is really gone
for n in root.iter('firstname'):
    print n.text
```

Next, we will look at ElementTree's `iterparse()` method. This method is very similar to the SAX method for parsing XML. There is one important difference. ElementTree will still build the document tree in the background as it is parsing. We can prevent this by clearing each element by calling its `clear()` method when are finished processing it.

```
f = et.iterparse('contacts.xml') # This is an iterator that you can use to go ↩
    forward and backward in the tree
for event, tag in f:
    print "{}: {}".format(tag.tag, tag.text)
    tag.clear()
```

We can also get both start and end events, however, start events are mostly useful for looking at attributes or to trigger some other action on element starts. The element is not guarenteed to be complete until the end event.

```
for event, tag in et.iterparse('contacts.xml', events=('start', 'end')):
    print "{} {}<{}>: {}".format(event, tag.tag, tag.attrib, tag.text)
```

Problem 3. Using ElementTree to parse `books.xml`, which represents a small dataset of books, and answer the following questions. Include the code you used with your answers.

1) Who is the author of the most expensive book in the list?

2) How many of the books were published before Dec 1, 2000?

3) Which books reference Microsoft in their descriptions?

HINT: To answer these queries, it may be most convenient to populate a pandas DataFrame with all the XML data.

Problem 4. The City of New York makes publicly available some data concerning the location of publicly available recycling bins. The XML endpoint is located at `https://data.cityofnewyork.us/api/views/sxx4-xhzg/rows.xml?accessType=DOWNLOAD`. Using the `requests` library, GET the XML file from that address and build it into an Element Tree.

Then, using Monte Carlo approximation, estimate the average distance from a given point in New York City to the nearest recycling bin. One efficient way of solving this problem is to perform a nearest neighbor search using a KDTree. We recommend you use scipy's `cKDTree` and its `query` method to find the nearest neighbor for each point.

Here's a quick example of how to use `cKDTree`.

```
>>> import numpy as np
>>> from scipy.spatial import cKDTree

>>> pts = np.random.rand(10,2)
>>> kdtree = cKDTree(pts)
>>> query_pts = [[0.5, 0.5], [0.25, 0.25]]
# k=1 corresponds to finding the nearest neighbor
>>> distance, index = kdtree.query(query_pts, k=1)
```

```
# 'distance' returns the distance to the nearest neighbor
# 'index' returns the index in 'pts' that corresponds to the
#     nearest neighbor
```

The `random_newyork_locations.csv` file contains about 35,000 uniformly distributed points throughout New York City measured in longitude and latitude.

We could measure the distance between the longitude and latitude points, however the units would not have a very interpretable and relatable value. Therefore, use the `convert()` function in Problem 2 to transform your longitude and latitude points to epsg:3857. These units very closely approximately meters.

Return your final answer in miles.

More information on this dataset is available at `https://catalog.data.gov/dataset/public-recycling-bins-eb48e`.
(Hint: If you get the file from requests, you may want to remove '<response>' and '</response>' from the beginning and end of your content string to make parsing a little easier.)

Additional Material

Pickle

Aside from the serialization methods we have demonstrated, Python has a serialization library called *pickle*. This library makes it very easy to serialize and share your python objects. The following code is a simple example of how to create a pickled object and then unpack it.

```
>>> import pickle

>>> listobject = [1, 2, 3, 4, 5]
>>> item = pickle.dumps(listobject)
>>> item
'(lp0\nI1\naI2\naI3\naI4\naI5\na.'

>>> pickle.loads(item)
[1, 2, 3, 4, 5]
```

You can also write these pickled objects to a text file as strings. The following code demonstrates this.

```

>>> a = open('list.txt', 'w')

>>> listobject = [1,2,3,4,5]
>>> item = pickle.dump(listobject, a)
>>> a.close()

>>> a = open('list.txt')
>>> pickle.load(a)
[1, 2, 3, 4, 5]
```

Pickle has many powerful applications such as these for communication between Python users. See `https://docs.python.org/2.7/library/pickle.html` for more documentation.

13 BeautifulSoup

Lab Objective: *Virtually everything rendered by an internet browser as a web page uses HTML. As a result, learning HTML is key to any kind of internet programming. BeautifulSoup is a Python package that helps navigate HTML documents. In this lab, we learn how to load HTML documents into BeautifulSoup and navigate the resulting BeautifulSoup object.*

HTML

HTML, or Hyper Text Markup Language is the standard markup language to create webpages. Just like XML, HTML tags describe different document content and are surrounded by angle brackets. Most tags can be combined with attributes such as `id` or `class` to help identify individual tags and make navigating the HTML tree much more simple. A list of all the current HTML tags can be found at `http://htmldog.com/reference/htmltags`. Here is an example:

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

The above example would output a single line

```
Click here for more information.
```

with 'here' being a clickable link to the website `http://www.example.com`.

While less readable, this HTML code can also be written as a single line as follows:

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

If a given tag doesn't contain any text or other tags, it can be written in a single pair of brackets as

```
<*tag_name*  ... *attributes*/>
```

The HTML of a website is very easy to view. Go to any website, such as `http://www.example.com`, in whatever browser is most convenient. Once on the website, right click with the mouse pointer and look for 'View Page Source' or a similarly worded option. Click it and the browser will open a new browser with the HTML code for your site. Some code is easy to follow, other code not so much.

Problem 1. Go to the website `http://www.example.com` and open the source code. What are all the tags used? What is the value of the `type` attribute associated with the `style` tag? Write a function that returns a list of all the tags used and the value of the `type` attribute.

Loading HTML into BeautifulSoup

Now that we know what HTML is, we can use BeautifulSoup to create a `BeautifulSoup` object. BeautifulSoup is a library capable of pulling data out of HTML scripts and files. BeautifulSoup works with a parser to provide commands to navigate and search the resulting HTML tree. Make sure the module `bs4` is installed in your Python packages. This section takes most of its material from `http://www.crummy.com/software/BeautifulSoup/bs4/doc/index.html`.

First we want a variable to store our HTML code as a string.

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

Next, we import `BeautifulSoup` from the `bs4` module. We call `BeautifulSoup()`, which takes as parameters the HTML string and an HTML parser. It returns a `BeautifulSoup` object, which represents the document as a nested data structure.

```
>>> from bs4 import BeautifulSoup
>>> soup = BeautifulSoup(doc, 'html.parser')
```

Including an HTML parser is optional, but a warning is given if one is not included. If that's the case, BeautifulSoup uses the HTML parser included in Python's standard library. Although other parsers are permitted, we have no need for them in our examples.

Once the document is stored, we can use `prettify()` to view the HTML. The `prettify()` method returns a string that can be printed to represent the BeautifulSoup object in a readable format. This will be useful later to make sure we are getting the correct HTML from websites.

```
>>> print(soup.prettify())
<html>
 <body>
  <p>
```

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

Problem 2. Write a function that loads the following string into BeautifulSoup, then prettifies it. Print the prettified string.

```
html_doc = """
<html><head><title>The Three Stooges</title></head>
<body>
<p class="title"><b>The Three Stooges</b></p>

<p class="story">Have you ever met the three stooges? Their names are
<a href="http://example.com/larry" class="stooge" id="link1">Larry</a>,
<a href="http://example.com/mo" class="stooge" id="link2">Mo</a> and
<a href="http://example.com/curly" class="stooge" id="link3">Curly</a>;
and they are really hilarious.</p>

<p class="story">...</p>
"""
```

> NOTE
>
> Note that the `<html>` and `<body>` tags are never actually closed. The parser used with `bs4` will automatically close these hanging tags, so don't get too stressed out by this example.

Navigating the HTML Tree

Since `BeautifulSoup()` returns an object which acts like a nested data structure, navigating it is very intuitive. We will use the following for the rest of the section, unless otherwise specified.

```
>>> soup = BeautifulSoup(html_doc, 'html.parser')
```

where `html_doc` is the document defined in problem 2.

By Tag Name

Because of the way BeautifulSoup objects store HTML tags, accessing them is as simple as calling the tag name. The output will be the called tag plus any nested tags or text. Below are some examples.

```
>>> soup.head
```

```
<head><title>The Three Stooges</title></head>

>>> soup.title
<title>The Three Stooges</title>
```

It is even possible to continue navigation down the tree through tags contained within tags.

```
>>> soup.body.b
<b>The Three Stooges</b>
```

Notice there are three `<a>` tags. When there are two or more tags of the same name, calling that tag only returns the *first* tag by that name.

```
>>> soup.a
<a class="stooge" href="http://example.com/larry" id="link1">Larry</a>
```

Tag Properties

In addition to viewing the entire tag, we can choose to access only certain properties. These properties include its name, attributes and strings (if applicable).

A tag's name is found using `.name`.

```
>>> tag = soup.a
>>> tag.name
u'a'
```

The attributes of a tag, if it has them, are stored in a dictionary and can be accessed as such. Accessing all the tags at once can be done through `.attrs`. Individual tags values can be reached by calling the key associated with it. If we try to access a key that is not an attribute, we get a `KeyError` in return.

```
>>> tag.attrs
{u'class': [u'stooge'], u'href': u'http://example.com/larry', u'id': u'link1'}

>>> tag['class']
[u'stooge']

>>> tag['href']
u'http://example.com/larry'

>>> tag['id']
u'link1'
```

Note that `class` returns a list. This is because the `class` attribute can have more than one value. This may show up in some HTML trees, but is not very common.

If a tag contains any text, it can accessed with `.string`.

```
>>> tag.string
u'Larry'
```

Problem 3. Using what you have just learned, write a function that returns the following from the Three Stooges example:

```
[u'title']
```

(Hint: This is the attribute in the eighth line of the prettified string from the previous problem.)

By Family Relations

Once we have selected a tag, we have several options available to navigate up, down, and sideways through an HTML tree.

Going Down

As mentioned before, a tag may contain other nested tags or text. These are called its children. Calling `.contents` returns the children of the parent tag in a list.

```
>>> head_tag = soup.head
>>> head_tag
<head><title>The Three Stooges</title></head>

>>> head_tag.contents
[<title>The Three Stooges</title>]

>>> title_tag = head_tag.contents[0]
>>> title_tag
<title>The Three Stooges</title>

>>> title_tag.contents
[u'The Three Stooges']
```

Note that the child of `title_tag` is the string `'The Three Stooges'`. Since strings cannot cannot have children, calling `.contents` on this string will return an error.

NOTE

One thing to note is the following:

```
>>> children_doc = """
...     <html><head>The Three Little Pigs</head>
...     <body>
```

```
...     <p>The first little piggy</p>
...     <p>The second little piggy</p>
...     <p>The third little piggy</p>
...     </body>
...     </html>"""
>>> pig_soup = BeautifulSoup(children_doc, 'html.parser')
>>> pig_soup.body.contents
[u'\n',
 <p>The first little piggy</p>,
 u'\n',
 <p>The second little piggy</p>,
 u'\n',
 <p>The third little piggy</p>,
 u'\n']
```

In this example, each new line character in the `<body>` tag is counted as a child of `<body>`. This will be very important when trying to navigate between *siblings*, or children of a common tag.

In addition to creating a list of children with `.contents`, we can use `.children` to create a generator of children tags. Using the previous example, we get the following (remember the extra carriage returns):

```
>>> for pig in pig_soup.body.children:
...     print(repr(pig)) #use repr() to ignore escape sequences
u'\n'
<p>The first little piggy</p>
u'\'n
<p>The second little piggy</p>
u'\n'
<p>The third little piggy</p>
u'\n'
```

There is a `.descendants` attribute which will recursively go through a tag's children, then the children's children, etc. It is left to the student to look at the online documentation for this attribute.

If a tag has only one child, and that child is a string, the child is available using `.string`. If a tag has one tag, and that tag has a single string as a child, then the parent tag can use `.string` to access the string as well.

```
>>> head_tag = soup.head
>>> print(head_tag)
<head><title>The Three Stooges</head></title>
>>> title_tag = head_tag.contents[0]
>>> print(title_tag)
<title>The Three Stooges</title>
>>>head_tag.string
u'The Three Stooges'
>>>title_tag.string
u'The Three Stooges'
```

If a tag contains more than one string, `.string` return `None`. However, `.strings` returns a generator that iterates through all strings contained within a tag. Check the online documentation for examples.

Going Up

Just as tags can have *childern*, tags can also have a *parent*. To access a tag's parent, we use the `.parent` attribute.

```
>>> title_tag = soup.title
>>> title_tag
<title>The Three Stooges</title>
>>> title_tag.parent
<head><title>The Three Stooges</title></head>
```

The parent of a string is the tag that contains it.

```
>>> tag = title_tag.string
>>> print(tag)
The Three Stooges

>>> tag.parent
<title>The Three Stooges</title>
```

Calling `.parents` iterates through all parents of a given tag. Examples can be found in the online documentation.

Going Sideways

Consider the following document, taken from the online documentation:

```
>>> sibling_soup = BeautifulSoup("<a><b>text 1</b><c>text 2</c></a>")
>>> print(sibling_soup.prettify())
<html>
 <body>
  <a>
   <b>
    text 1
   </b>
   <c>
    text 2
   </c>
  </a>
 </body>
</html>
```

Note the `<b>` and `<c>` tags are on the same level, underneath the `<a>` tag. These tags are considered *siblings*. Siblings in an HTML tree will always appear with the same indentation underneath a parent tag.

We use the attributes `.next_sibling` and `.previous_sibling` to navigate between these sibling elements. If a sibling has no next or previous sibling, calling these attributes returns `None`.

```
>>> sibling_soup.b
<b>text 1</b>

>>> sibling_soup.b.next_sibling
<c>text 2</c>

>>> sibling_soup.c.previous_sibling
<b>text 1</b>

>>> sibling_soup.c.next_sibling #<c> has no next sibling
None

>>> sibling_soup.b.string
u'text 1'

>>> print(sibling_soup.b.string.next_sibling) #text 1 and text 2 are not ↩
    siblings
None
```

Recall that in the `pig_soup` example we saw extra carriage returns between the `<p>` tags.

```
>>> pig_soup.body.contents
[u'\n',
 <p>The first little piggy</p>,
 u'\n',
 <p>The second little piggy</p>,
 u'\n',
 <p>The third little piggy</p>,
 u'\n']
```

What do you expect `pig_soup.body.p.next_sibling` to return?

```
>>> pig_soup.body.p.next_sibling
u'\n'

>>> pig_soup.body.p.next_sibling.next_sibling
<p>The second little piggy</p>
```

We need to make two calls to `.next_sibling` in order to get the next `<p>` tag. Keep this in mind for future questions as you navigate across siblings.

Just as with parents and children, there are also sibling generators `.next_siblings` and `.previous_siblings` to iterate through all the siblings of a given tag. These generators can be useful when multiple calls to `.next_sibling` must be made. As before, check the online documentation for more information.

Problem 4. Using the Three Stooges example and navigation by family relations, write a function that returns the following:

```
u'Mo'
```

Problem 5. Download the 'example.htm' file associated with the lab into your working directory (you can go to `http://example.com` to see the site this originates from). The following code opens and loads a file into a `BeautifulSoup` object:

```
>>> example_soup = BeautifulSoup(open('example.htm'), 'html.parser')
```

Write a function that returns the following line using two different methods.

```
u'More information...'
```

Have the function accept an integer `method`. If `method` is 1, return the line using your first method. If `method` is 2, return the line using the other.

By `find()`

In actual website HTML, often there are many tags that share a common name. It would be nice to find characteristics that might be unique for a given tag. Look back at our previous examples and think about what characteristics differentiate tags with the same name. BeautifulSoup uses `.find()` to allow you to search for a tag not only by name, but also by a specific attribute value or strings. The following examples refer back to the "Three Stooges" HTML document in problem 2.

Search by name:

```
>>> soup.find('b') #Pass in tag names, just like soup.b
<b>The Three Stooges</b>

>>> #or use the name parameter
>>> soup.find(name='a') #Still only returns the first instance
<a class="stooge" href="http://example.com/larry" id="link1">Larry</a>
```

Search by attribute:

```
>>> soup.find(id='link3') #Search by unique id attribute
<a class="stooge" href="http://example.com/curly" id="link3">Curly</a>

>>> #class is a Python keyword. Use 'class_' for the attribute key.
>>> soup.find(class_='title')
<p class="title"><b>The Three Stooges</b></p>

>>> #use the attrs parameter
```

```
>>> soup.find(attrs={'id':'link3'})
<a class="stooge" href="http://example.com/curly" id="link3">Curly</a>

>>> #combine attributes
>>> soup.find(attrs={'class':'stooge', 'href':'http://example.com/curly'})
<a class="stooge" href="http://example.com/curly" id="link3">Curly</a>

>>> soup.find(class_='stooge', href='http://example.com/curly')
<a class="stooge" href="http://example.com/curly" id="link3">Curly</a>
```

Search by string:

```
>>> soup.find(string='Mo') #Recall strings act as individual units
u'Mo'

>>> soup.find(string='Mo').parent #access the tag through the parent
<a class="stooge" href="http://example.com/mo" id="link2">Mo</a>
```

Search by combining parameters:

```
>>> soup.find('a', attrs={'id':'link2','class':'stooge'})
<a class="stooge" href="http://example.com/mo" id="link2">Mo</a>
```

Problem 6. Refer to the `example.htm` file. Load it using BeautifulSoup. Write a function that returns the tag associated with the "More information..." link using two different methods. At least one of these methods should use the `.find()` function. As before, have the function accept an integer `method`. If `method` is 1, use the first method. If `method` is 2, use the second method.

Problem 7. Download 'SanDiegoWeather.htm' and load it into BeautifulSoup. You can find the corresponding website at `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`. Using the `.find()` method, write a function that prints the tags referred to in the following questions:

1. What is the tag which contains the date, Thursday, January 1, 2015?
2. What are the tags which contain the links 'Previous Day' and 'Next Day'?
3. What is the tag which contains the number associated with the Actual Max Temperature?

(Hint: You can do a `ctrl+f` to find where the text is in the HTML, then study the tags around it.)

By `find_all()`

Recall that when a tag appeared multiple times, calling that tag name would return the first tag of that name.

```
>>> soup.a
<a class="stooge" href="http://example.com/larry" id="link1">Larry</a>
```

To get all instances of a certain tag, use the `find_all()` command.

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

This works with all the same arguments as the `.find()` function. You may refer to the online documentation for explicit examples.

Advanced Techniques

The following examples are techniques that can aid you in your search for specific tags. Consider the "Three Stooges" example from before. Suppose you want to find the tag that includes the url `http://example.com/curly`. You could search for it using the following:

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

This could be annoying to type out the whole website. Instead you could use regular expressions as follows:

```
>>> import re
>>> soup.find(href=re.compile('curly')) #find href containing 'curly' in it.
<a class="stooge" href="http://example.com/curly" id="link3">Curly</a>
```

This method can be used for tag names, attributes, and strings as well.

```
>>> soup.find(string=re.compile('Cu')).parent #find tag with string that starts↩
     with 'Cu'.
<a class="stooge" href="http://example.com/curly" id="link3">Curly</a>
```

To find a tag that has an attribute with a value, regardless of what the value is, we can use `True` and `False` in place of actual values. The following returns all tags that have some value associated with their `href` attribute:

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

Problem 8. Use BeautifulSoup to load the 'Big Data dates' file. This page can be found at `https://www.federalreserve.gov/releases/lbr/`. Note that the actual website may include more dates than the file provided. Notice all the release dates of bank data, ranging from 2003 to 2014 in the file you downloaded. Using `find_all()` and `re`, find all the links to bank data from September 30, 2003 to December 31, 2014. Write a function that loads the file into BeautifulSoup and returns a list of all of the tags containing these links.

14 Advanced BeautifulSoup

Lab Objective: *Learn how to use BeautifulSoup to scrape information from the internet and put it into easy-to-access data tables*

The internet is full of information. Sometimes this information is easy to read, sometimes it's not. No matter the case, web scraping is a useful tool used to transform unstructured data into structured data that is easier to analyze.

> ACHTUNG!
>
> Web scraping has legal implications. Do not scrape copyrighted information without the consent of the copyright owner. Many websites, in their terms and conmditions agreement, prohibit the practice of crawling parts or all of their website. Be careful and considerate when doing any sort of crawling. Be careful when writing and testing code so that there is no unintended behavior.

Scrapers and Crawlers

If you are on a website and all the information you want is contained on one page, you simply use a web scraper to read through the html code and pick out what you want. An example of this might be your professor's webpage, and you just need to scrape his phone number and email. Suppose instead that your information is found only after clicking through various links. For example, you want to find the email addresses and phone numbers for all the professors in the math deaprtment, but you can only get this information by clicking though an online directory, similar to `http://math.byu.edu/peopleresearch/faculty`. This would require *crawling* through various links to open up each professor's home page, and then scraping their sites. So scrapers are good for getting the information you need from a page, while crawlers will get you to the various pages from which you want to scrape information.

What Can You Scrape?

There are some website that permit the practice of web scraping. To ensure that scraping is well-behaved, most websites will tell a crawler where they can and cannot go, and scrapers what they can and cannot scrape. All of this information is included in a text file in the root domain of a website. The file is always titled `robots.txt` and defines considerate behaviors for web crawlers. Each robots file has a set of rules that label parts of a website as disallowed. Parts of a website that are not disallowed are implied to allow access by web crawlers. Many websites will limit crawlers to parts of the sites that will not place a large load on the website's server. It is your duty to honor the rules in `robots.txt` if they exist.

Simple Scraping

In lab 13, BeautifulSoup was used to read short bits of HTML code or a file using the `open()` command. Once the file is loaded, you can navigate through the HTML tree and pick out the data that you want.

Problem 1. Go to the url `http://www.federalreserve.gov/releases/lbr/20030930/default.htm`. Download the page source as an htm file and use BeautifulSoup to load the file. Using the methods taught, write a function that returns a 2-D NumPy array for bank information. Each row in the array represents a different bank, and each column represents a different piece of bank information. In your array, include the Bank Name, Rank, ID, Domestic Assets, and number of Domestic Branches for the following banks: JPMORGAN, CAPITAL ONE, and DISCOVER.

This is a very slow way to access information from a website. Remember this was only one in a list of over 20 different links to bank information. What if we wanted to get information from every link? Imagine trying to go through HTML trees for a hundred different websites only by copying and pasting HTML code!

Use the `urllib2` library in conjunction with BeautifulSoup to load HTML code from any website. We will go through some simple examples. First import `urllib2`. Use `urllib2`'s `urlopen()` function in conjunction with `read()` to load the HTML code into Python. Then simply use `BeautifulSoup()` to turn the HTML code into a navigable HTML tree. Note `urllib2` may need to be installed or updated before use.

```
>>> from bs4 import BeautifulSoup
>>> import urllib2

>>> url = "http://csb.stanford.edu/class/public/pages/sykes_webdesign/05_simple.html"

>>> content = urllib2.urlopen(url).read()
>>> soup = BeautifulSoup(content)

>>> print(soup.prettify())
<html>
 <head>
  <title>
```

```
   A very simple webpage
  </title>
  <basefront size="4">
  </basefront>
 </head>
 <body bgcolor="FFFFFF">
  ...
 </body>
</html>
```

Achtung!

Since `urllib2` accesses a website server, you may run into problems where you cannot establish a connection to the server. You can create a `try-except` clause to account for this, or just rerun your program. One possible way is as follows.

```
while True:
    try:
        content = urllib2.urlopen(url).read()
        break
    except:
        pass
```

Problem 2. Using the website `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` and BeautifulSoup, return the Actual Max Temperature. Return the tag which contains the link for the 'Next Day' button. Also, return the url attached to the link.

Sometimes, data we are interested in is contained over several different web urls. For example, what if we wanted a graph of the temperature highs over a period of time? For `www.wunderground.com`, the temperature history for a given city will be contained on separate pages for each day, just as in problem 2. In order to access information over several websites, we just need to load each new website into BeautifulSoup and locate the information we're interested in. Consider the following example.

In this example we look at the actual maximum temperature over the year 2014 in San Diego.

```
from bs4 import BeautifulSoup
import urllib2
import re

weather_url = 'https://www.wunderground.com/history/airport/KSAN/2014/1/1/↩
    DailyHistory.html'
```

```
weather_content = urllib2.urlopen(weather_url).read()
weather_soup = BeautifulSoup(weather_content)

actual = []

while('2015' not in weather_soup.find(class_='history-date').string):
    while(len(weather_soup.find_all(string='Actual')) != 1):
        weather_content = urllib2.urlopen(weather_url).read()
        weather_soup = BeautifulSoup(weather_content)
    actual_temp = weather_soup.find(string='Max Temperature').parent.parent.↩
        next_sibling.next_sibling.span.span.text
    actual.append(int(actual_temp))
    next_url = weather_soup.find(string=re.compile('Next Day')).parent['href']
    weather_url = 'https://www.wunderground.com'+next_url
    weather_content = urllib2.urlopen(weather_url).read()
    weather_soup = BeautifulSoup(weather_content)
```

Let's examine the code to see how it works.

After importing the necessary modules and opening up the url in BeautifulSoup, we define the variable `actual` to store the max temperatures in a list. Notice in the url that our dates start in 2014. Since we only want to go through the year 2014 and stop in 2015, we need a way to end our search once we get into 2015. Using `class_='history-date'`, we can find a tag which has the year in the string.

The next while loop is an error check. Sometimes this website does not load properly, so we check that all the information we need is located in the HTML code. In this case, we are looking for the actual max temp for a day, so we need to make sure the 'Actual' column shows up.

The next line of code defines `actual_temp`, which first directs us to the row of 'Max Temperature' and then navigates to the temperature column. This value is then turned to an `int` and stored in the list of temperatures.

Lastly, we find the url reference that is associated with the 'Next Day' link. We manually create the new url, keeping in mind that the new url found in the link is only an extension of the server website. This means that if the server is found at `http://www.wunderground.com` and the `'href'` value for the link is `/history/airport/KSAN/...`, then then new url we load into BeautifulSoup is `http://www.wunderground.com/history/airport/KSAN/...`. Therefore, we add the string representing the base url with the string representing the extension part of the url.

The output is the list of actual max temps over a year, which we can graph.

Problem 3. Adjust the above code to write a function that makes a list of the average max temperatures over a year. Graph this list, then return it. Note that because this code goes through a whole year's worth of urls, the function will take a long time to run.

Let's do another example before we turn you lose on the internet. This next example will look at the Commercial Bank data found at `http://www.federalreserve.gov/releases/lbr`. We will look primarily at the Consolidated Assets for JPMorgan over the years from 2003 to 2014.

```
from bs4 import BeautifulSoup
import urllib2
```

```
import re

bank_url = 'http://www.federalreserve.gov/releases/lbr/'
bank_content = urllib2.urlopen(bank_url).read()
bank_soup = BeautifulSoup(bank_content)

assets = []

dates_list = bank_soup.find_all(href=re.compile('((200[3-9])|(201[0-4])).*/default.htm'))

for url in dates_list:
    link_url = str('http://www.federalreserve.gov/releases/lbr/'+url['href'])
    link_content = urllib2.urlopen(link_url).read()
    link_soup = BeautifulSoup(link_content)
    amt = link_soup.find(string=re.compile('JPMORGAN')).parent.next_sibling.
        next_sibling.next_sibling.next_sibling.next_sibling.next_sibling.
        next_sibling.next_sibling.next_sibling.next_sibling
    assets.append(amt.string)
```

Now we will examine this code and see how it works.

We first import the necessary modules and load the url into BeautifulSoup. We create the variable `assets` to store the list of asset values.

To get the links we need, we use `find_all()` and select the unique identifiers of the link, namely that the links start with `20**` and end in `/default.htm`. Next we run the `for` loop to iterate through each link.

Just as in the previous example, we need to concatenate the server url `http://www.federalreserve.gov/release/lbr/` with the extension urls we stored in `dates_list`. This link is loaded into BeautifulSoup.

Now that we are on the webpage desired, we find the row we want by looking for the tag containing `'JPMORGAN'`. Once we have the tag, we navigate to the appropriate column. Notice we need to use 10 calls to `.next_sibling` in order to get to the correct information. The info is then appended into the data list.

Problem 4. Choose 1 of 3 options.

1. Load `http://www.google.com/finance` into BeautifulSoup. Towards the bottom of the web page, there is a Sector Summary. Go through each sector and locate the top five Gainers. In a SQL table, store the Name, abbreviation, % Change, and Mkt Cap of the top Gainer for each Sector.

2. Load `http://www.espn.go.com/nba/statistics` into BeautifulSoup. Go through the top five offensive leaders. In a SQL table, store the name, career games played, career mins per game, career points per game, and career FG% for each player.

3. Load `http://www.foxsports.com/soccer/united-states-women-team-stats` into BeautifulSoup. Go though each player on the World Cup US women's team. In a SQL table, store the name, hometown, position, and # of games played in the World Cup.

Advanced Scraping

The examples we have looked at so far have been very basic since the HTML is stored in the source code for the web pages. However, we will look at some examples where the HTML is written dynamically. This means the HTML is brought in from a separate source through javascript or ajax as a `.php` or `.aspx` table. These tables can be difficult to grab data from.

Go to the website `http://www.simplesoccerstats.com/stats/teamstats.php?lge=14&type=goals&season=0`. Open up the page source by right clicking and choosing the option for the page source. There is HTML code, but is it correct? Hit `ctrl+f` and search for 'Chicago,' one of the teams that appears on the actual webpage. It's not there! You can even try the following:

```
>>> from bs4 import BeautifulSoup
>>> import urllib2
>>> import re

>>> soccer_url = 'http://www.simplesoccerstats.com/stats/teamstats.php?lge=14&↩
    type=goals&season=0'
>>> soccer_content = urllib2.urlopen(soccer_url).read()
>>> soccer_soup = BeautifulSoup(soccer_content)

>>> print(soccer_soup.find(string='Chicago'))
None
```

Still nothing. This means the actual table of information is stored somewhere else.

Selenium

Selenium is a great tool to use on simple websites as well as websites with dynamic HTML source code. Basically Selenium will open up a browser and you can see what it is looking at. You can look at the source code of the actual website, and you can have some limited control over navigation, such as clicking links, clicking dropdown menus, pressing the back or forward buttons, etc. To use Selenium, import the following:

```
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
```

The `webdriver` allows you to use website functionality, while `Keys` allows you to use special keyboard keys like RETURN (i.e. when you want to send information through text boxes). Next, you want to open up a web browser and a website. For these exercises we will use Firefox. You can also use Chrome or IE if those are more familiar; all the commands will be similar.

```
driver = webdriver.Firefox()
example_url = 'http://www.example.com'
driver.get(example_url)
```

Note

If not already installed, you will have to download both Selenium and the driver associated with the browser you would like you use. More information and links to download can be found at `https://pypi.python.org/pypi/selenium`. In addition, the browser will need to be up to date.

The `.get()` command acts like `urllib2`'s `.urlopen()` and `.read()` combination. We can print out the HTML code using Selenium's `.page_source` attribute.

```
>>> print(driver.page_source)
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"><head>
    <title>Example Domain</title>
    ...
</body></html>
```

Once we have the source code, we can read it into BeautifulSoup.

Remember how we said Selenium reads HTML from a webpage differently than BeautifulSoup? Take a look again at the soccer example.

```
>>> from selenium import webdriver
>>> from bs4 import BeautifulSoup

>>> browser = webdriver.Firefox()
>>> soccer_url = "http://www.simplesoccerstats.com/stats/teamstats.php?lge=14&↩
    type=goals&season=0"
>>> browser.get(soccer_url)
>>> soccer_soup = BeautifulSoup(browser.page_source)
>>> browser.quit() #closes the web browser
>>> print(soccer_soup.find(string='Chicago').parent)
<td>Chicago</td>
```

This time there is a tag with 'Chicago' contained as text!

Problem 5. Consider the url `http://stats.nba.com/league/team/#!/?sort=W&dir=1`. Use Selenium to return a list of the `a` tags containing each of the 30 NBA teams. Use `.find_all()` in conjunction with whatever unique identifiers get you the correct tags. Hint: class and tag name are a good start.

Problem 6. Use the website from problem 5. Create a SQL table which stores the following information:

- The column titles are Name, HW%, AW%, where Name is each team name, HW% is the Home Win %, and AW% is the Away Win %.
- Each row represents a different basketball team, with its home and away win percentages.

Hint: You will need to use Selenium to access each teams website using the links from the tags found in problem 5. If the websites do not load properly, consider a `try-except` clause like the one suggested previously.

15 MongoDB

Lab Objective: *In this lab we introduce MongoDB, a non-relational database management system that is well suited to handling expanding datasets and dynamic storage. MongoDB communication is formatted almost entirely as JSON strings, and includes many of the same properties. We will use some of the more common MongoDB commands to investigate the architecture of a Mongo database and to explore similarities and differences between different documents.*

NoSQL Databases

Relational databases, such as SQL, were the most popular databases of the last decade. These databases rely on the data having relational attributes, meaning that each item in the database has the same attributes. We can visualize these databases as tables. As time passed and needs changed, relational databases became too structured for sets of data which involved unique attributes or were rapidly changing and expanding. In databases of this kind, each item may not have the same attributes. For example, in a database of wildlife, a salamander and an apple both have attributes of size and color, but an apple does not need a gender attribute and a salamander does not need a ripeness attribute. Relational databases store items with the same attributes, but if we want to store a salamander and an apple in the same database, we need a different type of database. Non-relational databases are built to allow for these types of databases and have opened the door to massively scalable databases that can store data in many forms and physical locations.

With these new databases, a new family of database managers arose to properly interface with them. Instead of designing a new relational database to meet every need, non-relational database managers were created that can adapt the different items for specific scenarios. MongoDB is such a manager. Several other managers, such as Cassandra, Redis, and Neo4j serve similar purposes. In this lab, we will focus on MongoDB.

MongoDB

MongoDB is a document database. It is best suited for storing data that does not have a fixed schema. Each MongoDB database is made up of collections of one or more documents. These documents are a special type of JSON object called BSON (Binary JSON). For the most part, BSON objects, JSON objects, and Python dictionaries can be used in much the same matter. However, there are a few subtle differences, such as with special characters. Trying to use a Python dictionary that contains the '$' character will often throw errors if it is used as though it were a BSON object.

MongoDB, being a database server manager, needs to have a location for the target database where it can store its data. Thus, we will need to run a database on a dedicated terminal window. If you do not have the MongoDB service running on your machine with a target database, please refer to the additional material section of this lab and follow the instructions to run an isolated test database server on your machine.

MongoDB has both a command line interface and Python bindings. This lab will use the official supported Python bindings, `Pymongo`. To install, you may use a package manager such as `pip` or download the binaries from the Pymongo website. More information for installation may be found at `http://api.mongodb.com/python/current/installation.html`.

After installation, Pymongo can be imported as with other standard libraries as follows:

```
from pymongo import MongoClient
'''
Create an instance of a client connected to a database running
at the default host ip and port.
'''
mc = MongoClient()
```

The following example illustrates a good use for MongoDB: Suppose you are running a general store. You have all sorts of inventory: food, clothing, tools, toys, etc. There are some attributes that every item has: name, price, and producer. Then there are attributes held by only some items: color, weight, gluten-freedom. A SQL database would have to be full of mostly-blank rows, which is extremely inefficient. More importantly, as you add new inventory, you will run across new attributes. With SQL, you would have to restructure and rebuild the whole database each time this happens. For MongoDB, this isn't a problem because it doesn't use the same relation tables. Instead, each item is a JSON-like object (similar to a Python dictionary), and thus can contain whatever attributes are relevant to the specific item, without including any meaningless attributes.

Creating and Removing Collections and Documents

A MongoDB database stores collections, and a collection stores documents. Each database can have several collections, each with its own documents. To visualize this, imagine we have a set of paper documents. We put the documents into folders (collections), and the folders into a filing cabinet (the database). When we need to add another collection, we simply create a reference to it and put it in the database. The new collection will not actually be created until we add documents to it, just as we would not file away a folder into the filing cabinet with all the rest until we have a document to be put into the folder. You can create a database and collection as follows:

```
# Create a new database
db = mc.db1

# Create a new collection
col = db.collection1
```

Documents in MongoDB are represented as JSON-like objects, and so do not adhere to a set schema. Each document can have its own fields.

```
col.insert({'name': 'Jack', 'age': 23})
```

```
col.insert({'name': 'Jack', 'age': 22, 'student': True, 'classes': ['Math', '↩
    Geography', 'English']})
x = col.insert({'name': 'Jill', 'age': 24, 'student': False})
```

We can check to see if the insert was successful by calling `x.is_valid(x)`.

Problem 1. Create a MongoDB database called `mydb` and a collection in `mydb` called `rest`. The file `restaurants.json` contains thousands of JSON objects, each describing a single restaurant. Load these into `rest`. The json.loads method should be helpful in doing this.

Querying for Documents

MongoDB uses a *query by example* paradigm for querying. This means that when you query, you provide an example that the database uses to match with other documents.

```
# Querying methods return a Cursor object which iterates through the result set↩
    .
r = col.find({'name': 'Jack'})
```

This query will return all documents in the collection that have the value 'Jack' in the 'name' field. You can also use the `count` method to return the number of documents that match your desired criteria.

```
# Find  how many 'students' are in the database
col.find({'student':True}).count()
```

We can update documents in a collection using `update`. Note that a simple update acts like a replace.

```
col.update({'name': 'Jack','student': True})
```

Problem 2. The file `mylans_bistro.json` contains a json object describing one additional restaurant. Insert it into the collection. Note that this entry contains an additional key value not present in any other. A SQL database would have to be entirely rebuilt to support this insertion, but with MongoDB this is not an issue.

After this insert, use a query to list every restaurant that closes at eighteen o'clock (Mylan's Bistro should be one of these).

Query Operators

There are several special operators that we can use to define conditions in a query. These query operators are used as keys and the queries are values.

Operator	Description
$lt, $gt	$<$, $>$
$lte	$\leq$, $\geq$
$in, $nin	Match any value in, not in an array, respectively
$or	Logical OR
$and	Logical AND
$not	Logical negation
$nor	Logical NOR (condition fails for all clauses)
$exists	Match documents with specific field
$type	Match documents with values of a specific type
$all	Match arrays that contain all queried elements

Table 15.1: MongoDB query operators

```
f = list(col.find({'age': {'$lt': 24}, 'classes': {'$in': ['Art', 'English']}})↩
    )
```

Problem 3. Query your new collection to answer the following questions:

- How many of the restaurants are in Manhattan?
- How many restaurants have gotten a grade other than an “A" on a health inspection?
- Which are the ten northernmost restaurants?
- Which restaurants have “grill" (case-insensitive) in their names?

Understand that MongoDB is not a relational database, therefore there is no concept of a join. This also means that we cannot define database relationships between documents. We can associate two documents by including a field that contains the unique ObjectID of the other document. When we request one document, we see it has an ObjectID, and then we run a second query to get the other document. Any “relational” things must be handled by the developer. This means that a document needs to contain all the information needed to find or retrieve it again.

Problem 4. Use update operators to perform the following tasks:

- Whenever a restaurant has “grill" in its name, replace “grill" with “Magical Fire Table".
- Increase all of the restaurant IDs by 1000.
- Delete the entries of every restaurant that has ever gotten a “C" health inspection grade.

Additional Material

Installation of MongoDB

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

Create Database Directory

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

Retrieve Shell Files

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

1. For Linux/Mac:

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

2. For Windows:

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

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

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