Contents

Lab 1

Unix Shell

Lab Objective: *Introduce the basics of the Unix Shell commands the Vim text editor*

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

The more you learn about 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.

File System

Navigation

Begin by opening the Terminal. The text you see in the upper left of the Terminal window is called the *prompt*. We will begin using three very basic commands: `pwd`, `ls`, and `cd`. The `pwd` command stands for **p**rint **w**orking **d**irectory. The `ls` command lists the files and subdirectories within the working directory. The `cd` command stands for **c**hange **d**irectory.

Problem 1. Using these commands, navigate to the `Shell-Lab` 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.

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

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

Flags

Most commands can be customized using flags. The `ls` command has dozens of optional flags. 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`. To view the reference manual for any command, use `man`. For example, to view the reference manual for the `ls` command, use `man ls`.

Other Useful Commands

Table 1.2 contains a list of commonly-used commands and their uses. Many of these commands will be needed throughout this lab and in any typical session with the Unix shell. After some of the commands are flags listed in square brackets that are worth exploring using `man`. We highly recommend experimenting with all these commands to become familiar with them. Remember you may freely experiment with these commands in the `test` directory.

```
$ 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
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
```

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`
`.`	Current directory
`..`	Parent directory
`~`	Home directory
`/`	Root directory

Table 1.2: Other useful commands dealing with the file system.

Problem 2. Inside the `Shell-Lab` 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, the naive way to do this would be to move each text file individually. However, this same result can be achieved using *wildcards*. We use wildcards as placeholder text. 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. See Table 1.3

Problem 3. Within the `Shell-Lab` 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 `Shell-Lab` directory. Do not move any of the files within these folders at this point.

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.

Pipes and Redirects

Unix becomes even more versatile and powerful when you chain multiple commands together. This is accomplished using *pipes*. Rather than printing the output of a command, the output is passed, or *piped*, to the next command. Two commands are piped together using the `|` operator. To demonstrate the power of pipes, we will first introduce a few 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 word count. 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 Shell-Lab/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, it is better to use *redirects*. 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.

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

Problem 4. 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.

Searching the File System

There are two powerful 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.

Problem 5. In addition to the .jpg files you have already moved into the `Photots` folder, there are a few other .jpg files in a few other folders within the `Shell-Lab` directory. Find where these files are using the `find` command and move them to the `Photos` folder.

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 by chance 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`.

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

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

```
$ zip zipfile.zip doc?.txt
 adding: doc1.txt (deflated 87%)
 adding: doc2.txt (deflated 90%)
 adding: doc3.txt (deflated 85%)

# 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   doc1.txt
     3634  2015-08-26 21:21   doc1.txt
---------                     -------
    16081                     3 files

$ unzip zipfile.zip
  inflating: doc1.txt
  inflating: doc2.txt
  inflating: doc3.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.

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

# 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

# use -x to extract
$ tar -zxvf <archive>
doc1.txt
doc2.txt
doc3.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. These text editors are accessed, as the name suggests, from the terminal. Vim is one of the most popular terminal text editors. For a beginner, the learning curve may be intimidating, but 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. There are countless keyboard shortcuts that allow you to navigate the file and execute commands without relying on a mouse, toolbars, or even the 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

We 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 editting the file. For our purposes, we want to create a new file.

You may notice if you start typing, the characters may or may not appear. 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. Now that we are in insert mode we may begin typing.

Problem 7. Create a new file in the `Documents` directory named `first_vim.txt`. Write at least multiple lines to this file. To return to command mode, hit the `Esc` key.

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

Insert mode should only be used for inserting text. It is not efficient to delete large portions of text while in insert mode. Try to get in the habit of leaving insert mode as soon as you are done adding the text you want to add.

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.

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

- 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
- g10 - **g**o to line **10**
- gg - beginning of file
- 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.

Command	Description
`dd`	delete line
`dl`	**d**elete **l**etter
`d4l`	**d**elete 4 **l**etters
`dw`	**d**elete **w**ord
`d2w`	**d**elete 2 **w**ords

Table 1.7: Commands for deleting in command mode

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. 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.

Deleting Text in Command Mode

As mentioned already, you should use insert mode only for adding new text. 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.

Quitting Vim

We quit vim by first enter last line mode. We do this by pressing the `:` key. When exiting vim, we most will want to save and quit or quit without saving. To save and quit, run `:wq`. To save without quiting, run `:q!`.

Problem 10. Save and exit the file you have created.

Customizing Vim

If you wish to customize vim commands, this is accomplished using the `:map` command. For example, if you plan on using vim extensively, we highly recommend you remap `Esc` to a more convenient key sequence like `jk`. This would be accomplished by running `:map jk <Esc>`. You can save these customizations in the `vimrc` file.

Command	Description
`: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

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.

Lab 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 we built in 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 whatever else may be necessary to learn in the future.

File Security

To begin, run the following command while inside the `Shell-Lab` directory:

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

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

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

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

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

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"
```

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

$ chmod u+x hello
$ ./hello
Hello World
```

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

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

Table 2.3: Other commands when working with permissions

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

Table 2.4: Commands for resource management

Problem 1. In the `Python` directory you will find `count_files.py`. `count_files.py` is a python script that counts all the files within the `Shell-Lab` 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.

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

Resource Management

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

Job Control

Let's say we had a series of scripts we wanted to run. If we knew that these would take a while to execute, we may want to start them all at the same time and let them run while we are working on something else. In the 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.

```
$ ./ ten_secs &
$ ./ five_secs &
$ jobs
[1]+  Running       ./ten_secs &
```

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

```
[2]-  Running       ./five_secs &
$ kill %2
[2]-  Terminated    ./five_secs &
$ jobs
[1]+  Running       ./ten_secs &
```

Problem 2. 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',
 '']
# 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]
```

Problem 3. Create a `Shell` class in Python. Write a `find_file` method that will search for a filename starting in the current directory using the `find` command. Write a `find_word` method that finds a given word within the contents of the current directory using the `grep` command. For both these functions, return a list of filepaths.

Problem 4. 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.

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.

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.

```
$ 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
```

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 5. 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

In many of its applications, `wget` and `curl` perform the same tasks. Both of these commands are used to download content from the internet. Most 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 and `wget` can download recursively. Though we will provide examples using `wget`, know that much of the same functionality can be performed using `curl`.

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 6. 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.

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 ¡url¿

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`. Whenever we are wanting to call a command using `awk`, we always 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 examples we will be exploring in this lab, 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 ¡url¿. 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. In the `Documents` directory, you will find a `people.txt` file that we will use for the following examples.

```
# 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 7. 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`.

One Final Note

Though there are multiple Unix shells, one of the most popular is the *bash* shell. The bash shell is highly customizeable. In your home directory, you will find a hidden file named `.bashrc`. All customization changes are saved in this file. If you

are interested in customizing your shell, you can customize the prompt using the `PS1` environment variable. As you become more and more familiar with the Unix shell, you will come to find there are commands you run over and over again. You can save commands you use frequently using `alias`. If you would like more information on these and other ways to customize the shell, you can find many quality reference guides and tutorials on the internet.

Lab 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.

> **Warning**
>
> 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.

Warning

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.

Problem 6. A *valid python parameter definition* is defined as the concatenation of the following strings:

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

Define a variable `parameter_pattern_string` that defines a regular expression that matches valid python parameter definitions.

For example, each element of `["max=4.2", "string= ''", "num_guesses", "msg ='\\''", "volume_fn=_CALC_VOLUME"]` is a valid python parameter definition, while each element of `["300", "no spaces", "is_4=(value==4)", "pattern = r'^one|two fish$'", 'string="these last two are actually valid in python, but they should not be matched by your pattern"']` is not.

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.

Warning

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`.

Lab 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 are 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.

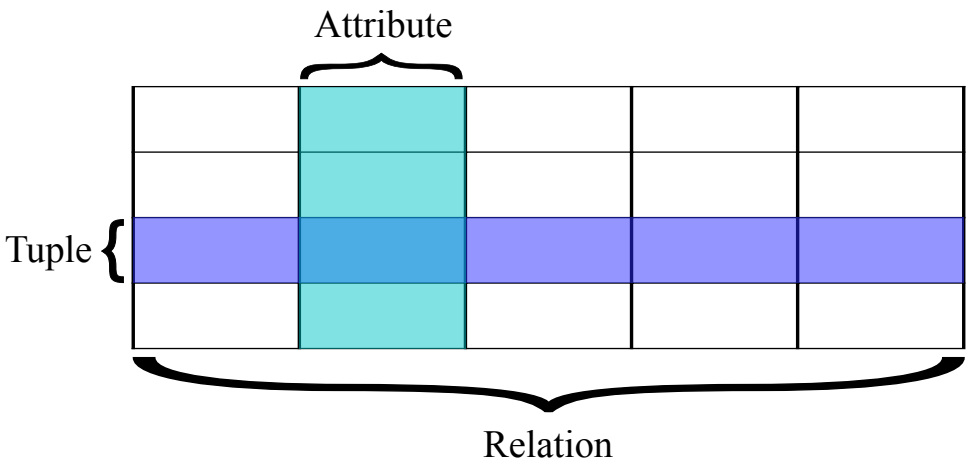

Figure 4.1: Elements of a relation.

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`

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:

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

```
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.

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.

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

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

The cursor object has several useful methods (Table 4.3).

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.

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

Creating and Dropping Tables

Let's create a table.

```
cur.execute('CREATE TABLE student_information (StudentID INTEGER NOT NULL, Name ↩
    TEXT, SSN INTEGER, 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.

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.

StudentID	Name	SSN	MajorCode

Table 4.5: student_information

StudentID	Name	SSN	MajorCode
55	John Smith	372897382	2

Table 4.6: student_information

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

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

Problem 1. In this problem you will create two new tables. The first table will be called MajorInfo and have a column called MajorCode and MajorName. MajorCode 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.

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 student_information VALUES(55, 'John Smith', 372897382, ↩
    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.

```
cur.execute("INSERT INTO student_information(MajorCode, Name, SSN, StudentID)  ↩
    VALUES(55, 'John Smith', 372897382, 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 student_information 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 student_information WHERE MajorCode=55;")
```

Warning

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

Updating Rows of a Table

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

```
cur.execute("UPDATE student_information 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.

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. Yourn 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=',')]
```

Problem 3. Create the following tables in the same database you created in Problem 2. 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: students

ID	Name
1	Math
2	Science
3	Writing
4	Art

Table 4.8: majors

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: grades

ClassID	Name
1	Calculus
2	English
3	Pottery
4	History

Table 4.10: classes

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).

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

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 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.

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
```

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?

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.

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.

Lab 5

Advanced SQL

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.

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

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 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.

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.

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

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.

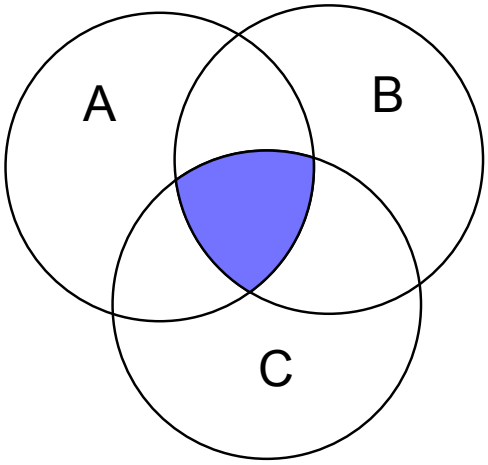

Figure 5.1: An inner joining of tables A, B, and C.

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.

```
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 performs a left outer join is

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

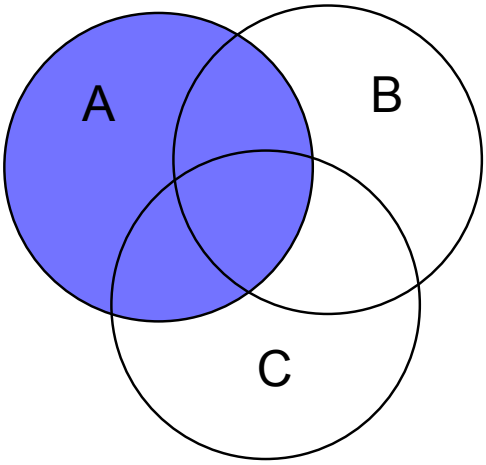

Figure 5.2: A left outer table join of A with tables B and C.

```
for row_s in students:
    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;
```

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.

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

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

Advanced Selections

Aggregate functions are useful for summarizing the data in a column. The functions are

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.

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.

Problem 2. Write a SQL query that will 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

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 why 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.

```
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 and 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.

Problem 5. 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.

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

Lab 6

Intro to pandas I

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 like R and Stata. We now turn our attention to *pandas*, a Python library that is more specifically built for data analysis.

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.

Series

The `Series` is a one-dimensional array with labeled entries. The values of the array may be any data type, including integers, strings, or general Python objects. Further, the array need not be homogeneous. That is, it can hold values of different data types. Together, the array 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(range(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=range(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 when creating 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. 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, 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
```

Warning

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=range(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 `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, but not strictly necessary for most applications. The interested reader is invited to explore the documentation to learn more.

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 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 access, plotting, 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` methods. The `.loc` method 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   -0.171938
d   -0.814336
Name: I, dtype: float64
```

Finally, a column of a `DataFrame` may be accessed using simple square brackets and the name of the column:

```
>>> # get second column of df
>>> 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
```

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.

We start by plotting 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.

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.

Using `DataFrames`, one can also plot one column against another.

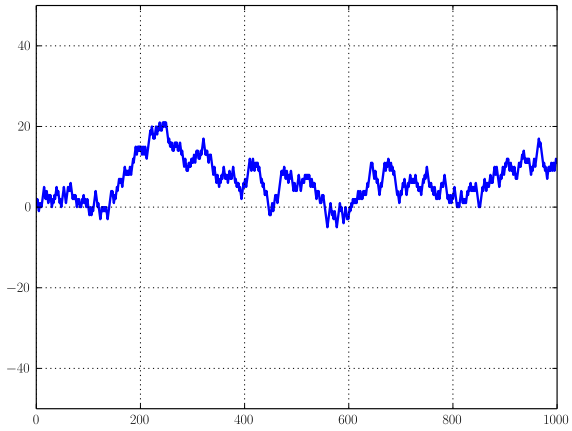

Figure 6.1: Random walk of length 1000.

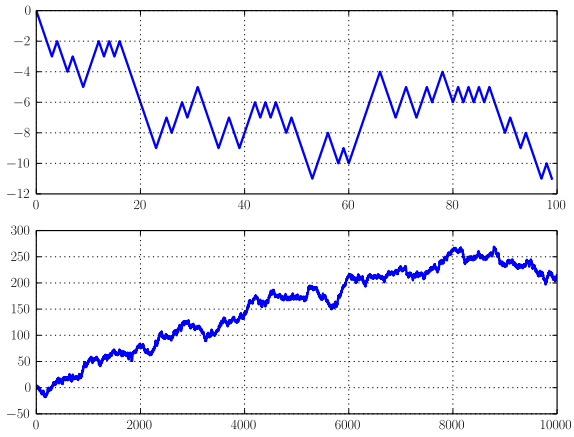

Figure 6.2: Biased random walk of length 100 (above) and 10000 (below).

```
>>> xvals = pd.Series(np.sqrt(np.arange(1000)))
>>> yvals = pd.Series(np.random.randn(1000).cumsum())
>>> df = pd.DataFrame({'xvals': xvals, 'yvals': yvals})
>>> df.plot(x='xvals', y='yvals') # specify x and y values
>>> plt.show()
>>> plt.close()
```

The result is displayed in Figure 6.3.

A variety of other types of plots are possible. One of the more useful plots when trying to estimate or visualize the distribution of data is a histogram. The code listed below demonstrates how to generate a histogram for each column in a `DataFrame`, with the result shown in Figure 6.4.

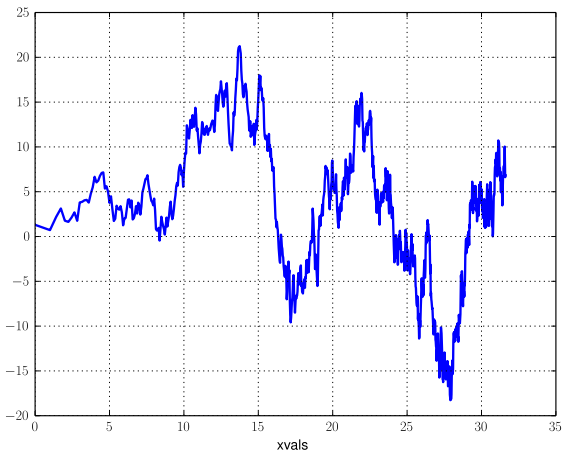

Figure 6.3: Graph generated when one coordinate is taken from the xvals column and the other from the yvals column.

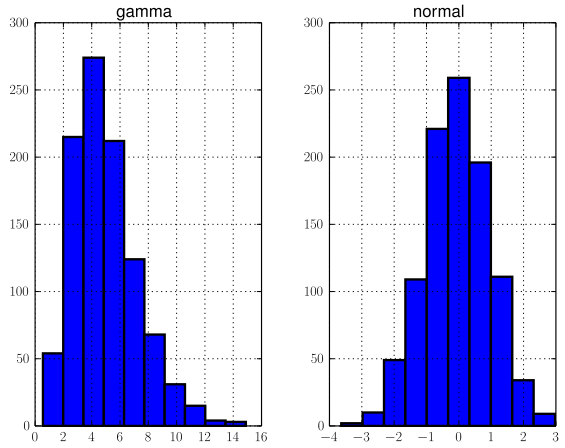

Figure 6.4: Histogram of two columns of a `DataFrame`.

```
>>> col1 = pd.Series(np.random.randn(1000))         #normal distribution
>>> col2 = pd.Series(np.random.gamma(5, size=1000)) #gamma distribution
>>> df = pd.DataFrame({'normal': col1, 'gamma': col2})
>>> df.hist()
>>> plt.show()
>>> plt.close()
```

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 this data:

```
>>> #build toy data for SQL operations
>>> name = ['Bill', 'Alice', 'Joe', 'Jenny', 'Ted', 'Taylor', 'Tracy', 'Morgan', ↩
    'Liz']
>>> 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)
```

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 Math_Major, COUNT(*) FROM mathInfo GROUP BY Math_Major
>>> print mathInfo.groupby('Math_Major').size()
```

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    Bill   M    4.0          y
1   21    Se   1   Alice   F    3.0          n
2   22    Se   3   Jenny   F    4.0          n
3   20     J   5  Taylor   F    3.5          y
4   20     J   6   Tracy   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(columns='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`.

Lab 7

Intro to pandas II

In this lab, we explore in further detail two specific areas where pandas can be a very useful tool: analyzing sequential data, and working with large datasets that can't be stored entirely in memory.

Warning

This lab assumes pandas 0.14.0. Errors may occur if you have an earlier version. You can check the version of pandas you are running by the following:

```
>>> import pandas as pd
>>> pd.__version__
'0.14.0'
```

Time Series Analysis

A *time series* is a particular type of data set that consists of a sequence of measurements or observations generated at successive points in time. Examples include the yearly average temperate of a city, or the price of a given stock measured daily.

Working With Large Datasets

In the real world, a data scientist is often confronted with large datasets that can't be held in memory all at once. There are various solutions to this problem; in this section, we will explore how pandas uses the HDF5 file format to allow us to work with datasets on disk.

HDF5, which stands for "Hierarchical Data Format", is a data storage system especially suited for large numerical datasets. Rich and efficient software libraries have been developed over the years to enable fast read and write operations, which make HDF5 a competitive option for working with large datasets in many applications. The Python library pytables is one such library, and the HDF5 capabilities in pandas are built directly on top of pytables.

We have two primary learning goals: how to get our data into the proper HDF5 format, and how to intelligently work with the data once it's tidied up. Let's dive in.

Writing HDF5 Data

The primary way we will interact with HDF5 data goes through the `HDFStore` object, which behaves somewhat like a dictionary. To begin, let's instantiate such an object, and write data to it. Make sure to execute the code snippets throughout to ensure that everything works as expected on your machine.

```
>>> # we will create an HDFStore with the filename test_store.h5
>>> my_store = pd.HDFStore('test_store.h5')
>>> my_store
<class 'pandas.io.pytables.HDFStore'>
File path: test_store.h5
Empty
```

The file `'test_store.h5'` has just been created in your working directory, although it contains nothing as yet. Write some pandas data to the store:

```
>>> # instantiate data, write to store
>>> ts = pd.Series(index=['A', 'B', 'C', 'D'], data = np.random.randn(4))
>>> df = pd.DataFrame(index=range(6), columns=['a','b'], data=np.random.random↩
    ((6,2)))
>>> my_store['ts'] = ts
>>> my_store['df'] = df
>>> my_store
<class 'pandas.io.pytables.HDFStore'>
File path: test_store.h5
/df            frame        (shape->[6,2])
/ts            series       (shape->[4])
```

Note the dict-like syntax for writing data to the store. We now see that the store contains two objects, which can easily be retrieved in the following manner:

```
>>> # retrieve the df from the store, check it is the same
>>> store_df = my_store['df']
>>> (df == store_df).all()
a    True
b    True
dtype: bool
```

Removing an object that you have written to the store can be accomplished as follows (although note that removing the object doesn't necessarily free up space on the hard disk, so the file size may not decrease):

```
>>> # let's remove the series ts
>>> del my_store['ts'] # or my_store.remove('ts')
>>> my_store
<class 'pandas.io.pytables.HDFStore'>
File path: test_store.h5
/df            frame        (shape->[6,2])
```

The read and write operations that we have explored so far work just fine for small objects that can fit entirely in memory, but the story is different when it comes to writing large datasets to an HDF5 file. Suppose we have a CSV file containing the large dataset that we want to work with. One basic approach to storing this data in HDF5 format is to read and write it by *chunks*, that is, move the data into an HDF5 store a few lines at a time. This ensures that we never have to read too much of the data into memory at once.

The `read_csv` function in pandas allows for reading a file by chunks of rows. We simply need to specify the keyword argument `chunksize`, which gives the number of rows of the file to read in each time. Most other pandas data readers have a similar option for loading data by chunks. It is also important to specify the correct data types of each column when reading in the chunks of data. This will ensure that a consistent data type is used when writing to the HDF5 store. To do this, create a dict that maps each column name to its correct data type. Columns containing strings should have the `object` datatype, and columns containing numerical data may be ints or floats. Any column that contains date data should NOT be included in the dictionary, but rather should be passed as the `parse_dates` keyword argument (a list of the column names containing dates).

To iteratively write data to a single object in an HDF5 store, we must use the `append` method. This will store the data in a particular format called a *table*, an on-disk data structure geared toward efficient querying of the rows. There are a few parameters that we must tune in order to write the data successfully.

- `key`: This is the target object in the HDF5 store to which we want to write.
- `value`: This is the actual chunk of data (such as a `DataFrame`) that we want to append to the store.
- `data_columns`: A list of the column names of the incoming data that should be indexed. You should include all columns that you will use in subsequent queries of the data. For example, if my dataset has a column labeled 'A', and I anticipate wanting to select all rows where the value in column 'A' is greater than, say, 0, then it is imperative that `data_columns` includes 'A'. Try to avoid including columns that aren't needed in the queries, as performance can decrease with a larger number of indexed columns. This argument only needs to be specified for the *first* chunk to be written.
- `min_itemsize`: This is an int that specifies the maximum length of a string found in the dataset. If you are unsure of the exact length of the longest string, try setting this to a high default value (say 100). If you attempt to write data containing a string whose length exceeds `min_itemsize`, an error is raised. This argument is also only required for the first chunk.
- `index`: This argument is a boolean flag that indicates whether the data should be indexed as it is written. By setting `index=False`, the data is written much faster.

In the code below, we create a CSV file containing toy data, then write it by chunks to our HDF5 store.

```
>>> # first create the toy CSV file
>>> n_rows = 10000
>>> n_cols = 3
>>> csv_path = 'toy_data.csv'
>>> csv_file = open(csv_path, 'w')
>>> csv_file.write("A B C\n")
>>> for i in xrange(n_rows):
>>>     for num in np.random.randn(3):
>>>         csv_file.write(' ' + str(num))
>>>     csv_file.write('\n')
>>> csv_file.close()

>>> # now iteratively write the data to the store
>>> n_chunk = 1000
>>> col_types = {'A':np.float, 'B':np.float, 'C':np.float}
>>> reader = pd.read_csv(csv_path, sep=' ', dtype=col_types, index_col=False, ↩
    chunksize=n_chunk, skipinitialspace=True)
>>> first = True # a flag for the very first chunk
>>> data_cols=['B', 'C'] # queries involving cols B and C will be allowed
>>> for chunk in reader:
>>>     if first:
>>>         my_store.append('toy_data', chunk, data_columns=data_cols, index=↩
    False)
>>>         first = False
>>>     else:
>>>         my_store.append('toy_data', chunk, index=False)
>>> my_store
<class 'pandas.io.pytables.HDFStore'>
File path: test_store.h5
/df                 frame        (shape->[6,2])
/toy_data           frame_table  (typ->appendable,nrows->10000,ncols->3,indexers↩
    ->[index],dc->[B,C])
```

If an error of any type occurs when writing your data, and you need to start over, it is necessary to remove the previously written data, since the `append` function doesn't overwrite existing data.

Now that the data is in the HDF5 store, we should index the table, which can speed up query operations. This is done as follows:

```
>>> my_store.create_table_index('toy_data', optlevel=9, kind='full')
```

Obviously the toy data in this example is small enough to fit in memory, but it illustrates a basic approach to moving large datasets into a HDF5 store.

Problem 1. Write the data contained in the file `campaign.csv` to an HDF5 store using the chunking approach. We recommend setting the `chunksize` argument to 50,000. There will be just over 100 chunks for this chunk size, so you can track your progress by printing out a counter, if you wish. The whole writing process will likely take a few minutes.

The file `campaign_format.txt` contains information about the dataset, including the column names and descriptions. Consult this file and note which, if any, columns contain date information. Use this information to appropri-

ately set the `parse_dates` argument in the `read_csv` function. This argument should be a list of the column names that include date information. Note also that some of the columns contain strings, so remember the `min_itemsize` argument. Finally, you will be analyzing this data in the remainder of the lab, so glance over the problems below to determine which columns need to be indexed.

Once you have finished writing to the store, create a table index for the data, just as shown in the example above.

Working With On-Disk Arrays

Now that we have the data in a HDF5 table, how do we work with it? The key here is to only read in subsets of the data, since trying to read the entire dataset at once may result in swamping the memory and crashing the system. The `select` method allows us to do just this. It takes two basic arguments: first, the name of the table in the store that we want to query, and second, a query statement. Remember, the query statement may not reference columns that weren't included in `data_columns`.

```
>>> # this query is OK
>>> my_store.select('toy_data', where = ["'B' < 0", "'C' > 0"])

>>> # this query is NOT OK
>>> my_store.select('toy_data', where = ["'A' < .5"])
```

Note that each logical statement in the `where` list is combined with a logical AND.

Let's look at some simple examples with our campaign dataset. We assume that the data is in a HDF5 store called `store`. Follow along to make sure you get the same results.

```
>>> # get a list of the candidates; result should be 14 candidates
>>> cands = store.select_column('campaign', 'cand_nm').unique()

>>> # find number of contributions from UT
>>> n_UT = len(store.select('campaign',where = ["'contbr_st' == 'UT'", "columns='↩
    contbr_st'"]))
>>> n_UT
50462

>>> # find number of UT contributions to Romney
>>>  n_rom_UT = len(store.select('campaign', where = ["'contbr_st' == 'UT'", "'↩
    cand_nm'=='Romney, Mitt'", "columns='contbr_st'"]))
>>> n_rom_UT
26962

>>> # proportion of UT contributions that went to Romney
>>> np.float(n_rom_UT)/n_UT
0.534303039911
```

Problem 2. How many contributors in California gave to Obama? To Romney?

It is also possible to execute more complicated queries involving grouping operations, although some care is required. Consider the following question: how many contributions came from each state, and what is the average contribution from each state? To answer this question, we need to aggregate information grouped by each state. If we could hold the data in memory, we could use a simply `groupby` operation, but since our data resides on disk, the solution is a bit more involved. Fortunately, the `'contbr_st'` column alone is small enough to load into memory, so we do that. We can then utilize the `Series` method `value_counts`, which counts the number of occurrences of each distinct value in the column. In this way, we obtain a `Series` indexed by the states, and containing the number of contributions from each state.

```
>>> # load in the state column, then calculate the counts
>>> states = store.select_column('campaign','contbr_st')
>>> states = states.value_counts()
```

Next, we want to iterate through each state, and calculate the mean contribution.

```
>>> st_contb = []
>>> for state in states.index:
>>>     grp = store.select('campaign', where=["'contbr_st'=\"{}\"".format(state),↩
    "columns='contb_receipt_amt'"])
>>>     st_contb.append(grp['contb_receipt_amt'].mean())
```

To tidy up our results, we create a `DataFrame` indexed by the state, containing the number of contributions and average contribution. Then we plot top results.

```
>>> state_info = pd.DataFrame({'Count':states, 'Avg Contb':st_contb})
>>> state_info.sort(columns='Count', ascending=False, inplace=True)
>>> plt.subplot(211)
>>> state_info['Count'][:10].plot(kind='bar')
>>> plt.subplot(212)
>>> state_info['Avg Contb'][:10].plot(kind='bar')
>>> plt.show()
```

Results are shown in Figure 7.1

Problem 3. Calculate the total net contributions to each candidate, and plot the results in a bar graph. Your result should match Figure 7.2.

Problem 4. Calculate the frequency of the 20 most common occupations of contributors in the dataset. Also calculate the average positive contribution

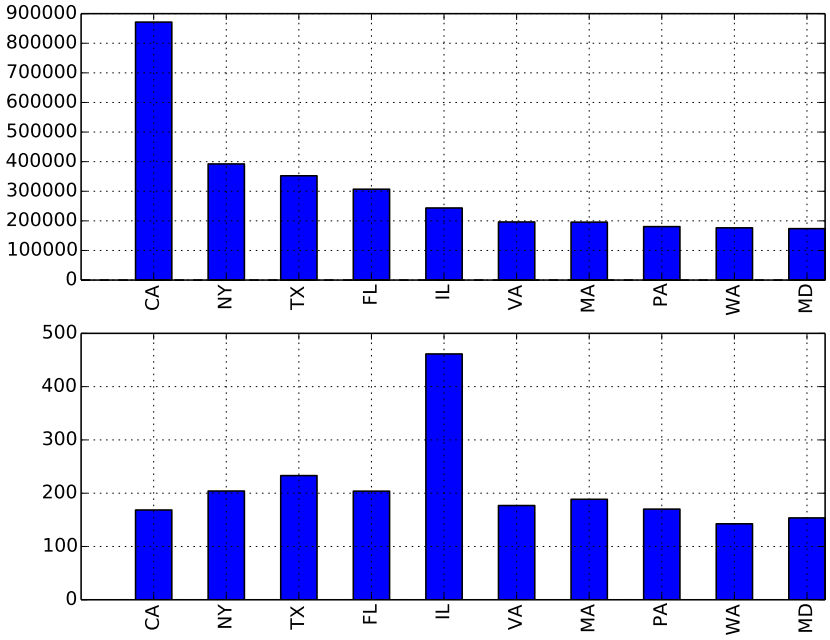

Figure 7.1: Top 10 contributing states (top), and their average contributions (bottom).

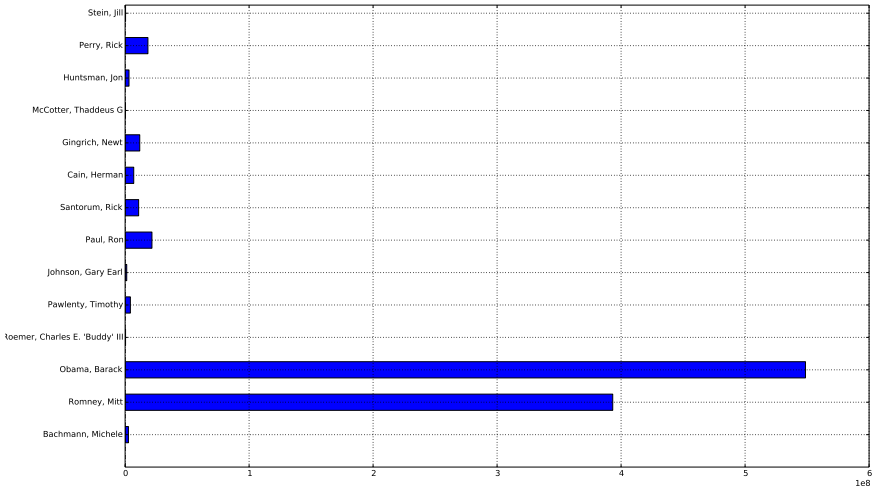

Figure 7.2: Total net contributions to each candidate.

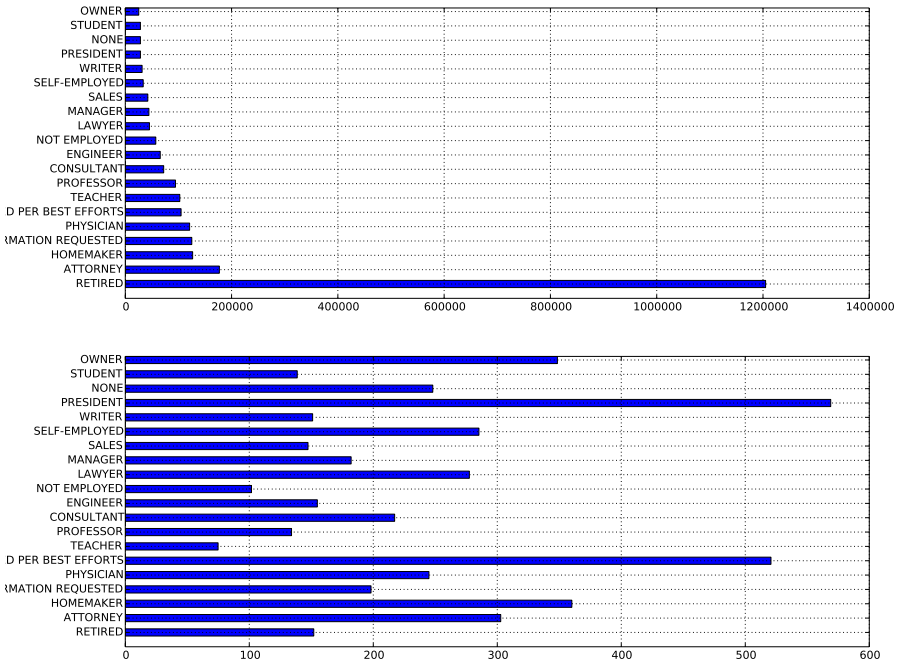

Figure 7.3: Top 20 occupations of contributors (above) and average contribution of each occupation (below).

amount for each of these 20 occupations. Plot the results in two bar graphs. You results should match Figure 7.3.

What if you are interested in the contributions to a candidate as a function of time? Let's first aim to create a `DataFrame` containing the contribution amount and date for all contributions going to Ron Paul.

```
>>> cand = 'Paul, Ron'
>>> cand_contb = store.select('campaign', where=["'cand_nm'==\"{}\"".format(cand)↩
    ])[['contb_receipt_amt', 'contb_receipt_dt']]
```

Next, we need to group by date, and apply the sum function to add up contributions for each particular date. This can be done with the `groupby` function as follows:

```
>>> contb = cand_contb.groupby(by='contb_receipt_dt').sum()
```

Plotting this time series yields Figure 7.4.

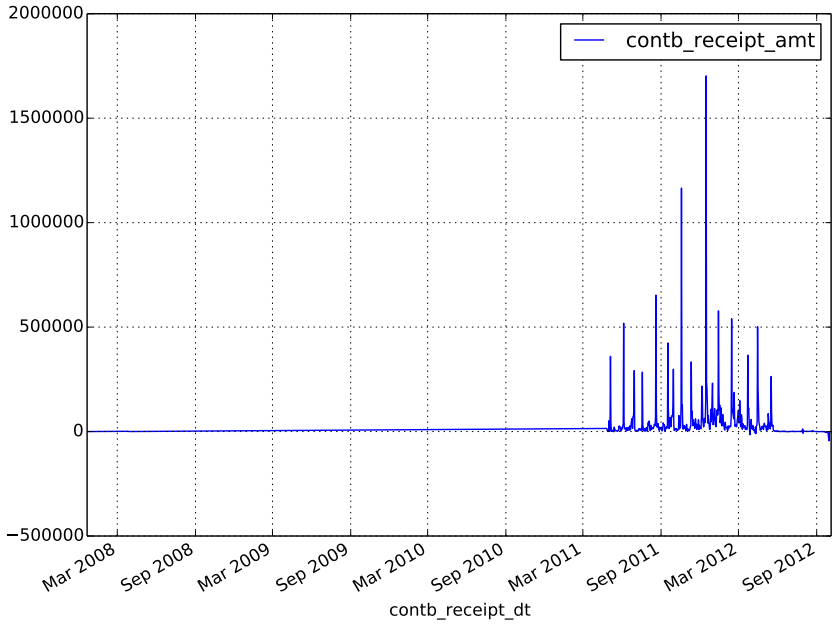

Figure 7.4: Campaign Contributions to Ron Paul over time.

Problem 5. Plot the running total of campaign contributions as a function of time for Mitt Romney, Barack Obama, and Newt Gingrich (all on the same graph). Your results should match Figure 7.5.

Each data project brings with it a new set of problems and pitfalls, but a careful application of the principles in this lab will provide a good starting point for working with large datasets in pandas.

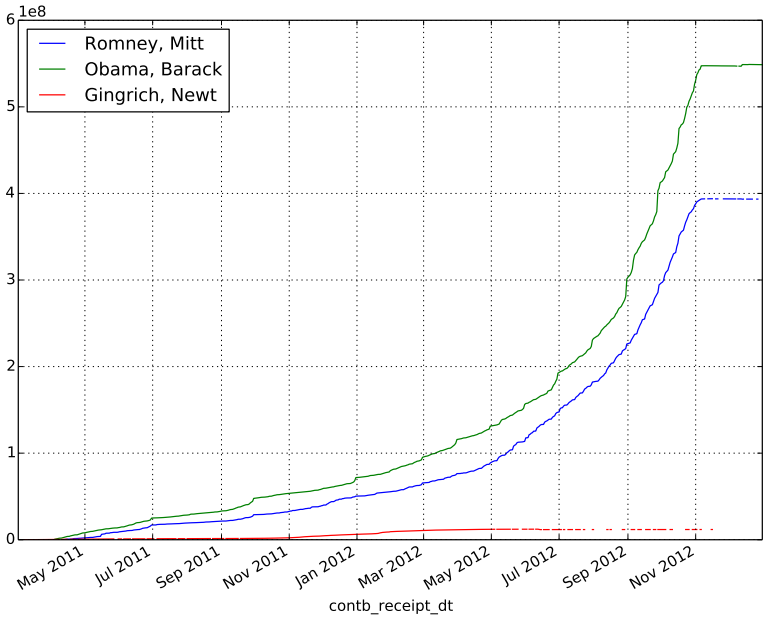

Figure 7.5: Running contribution totals for three candidates.

Lab 8

Web Technologies

Lab Objective: *Learn about serialization and markup languages. You will also learn how to communicate using the HTTP protocol.*

Since the dawn of computing, the ability to make computers talk to each other has captivated the minds of many. It wasn't until the 1960s that such capabilities were explored in depth. Today's world would not be possible if computers didn't have the capability to network. The ability to send and receive rich, meaningful data is the focus of our exercises in this lab.

Serialization

How would you store a Python list or dictionary outside of the interpreter? Suppose we calculated some results and stored them in a list that we wanted to send to a friend or colleague? However we choose to store our list, we need to be able to load it back into the Python interpreter and use it as a list. What if we wanted to store more complex objects? The process of serialization seeks to address this situation. Serialization is the process of storing an object and its properties in a form that can be saved or transmitted and later reconstructed back into an identical copy of the original object.

JSON

JSON, pronounced "Jason", stands for *JavaScript Object Notation*. This serialization method stores information about the objects as a specially formatted string. It is easy for both humans and machines to read and write the format. When JSON is deserialized, the 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. These data structures are more familiarly called dictionaries and lists in Python. Python's standard library has a module that can read and write JSON. Most JSON libraries, though, have a fairly standard interface. If performance is critical, there are Python modules for JSON that are written in C such as ujson and simplejson.

Let's begin with an example.

```
>>> import json
>>> json.dumps(range(5))
'[0, 1, 2, 3, 4]'
>>> json.dumps({'a': 34, 'b': 483, 'c':"Hello JSON"})
'{"a": 34, "c": "Hello JSON", "b": 483}'
```

As you can see, the JSON representation of a Python list and dictionary are very similar to their respective string representations. You can also see that each JSON message is enclosed in a pair of curly braces. We can even nest multiple messages.

```
>>> a = """{"car": {
    "make": "Ford",
    "model": "Focus",
    "year": 2010,
    "color": [255, 30, 30]
        }
    }"""
>>> t = json.loads(a)
>>> 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]
```

Most JSON libraries support the dump[s]/load[s] interface. To generate a JSON message, we use `dump` which will accept the Python object and generate the message and write it to a file. `dumps` does the same, but just returns the string rather than writing it to a file. To perform the inverse operation, we use `load` or `loads` for reading from a file or string respectively.

Many websites and web APIs make extensive use of JSON. Twitter, for example, return JSON messages for all queries.

The built-in JSON encoder/decoder only has support for the basic Python data structures such as lists and dictionaries. Trying to serialize a set will result in an error

```
>>> a = set('abcdefg')
>>> json.dumps(a)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-373a2a7edfd2> in <module>()
----> 1 json.dumps(a)

/opt/anaconda/lib/python2.7/json/__init__.pyc in dumps(obj, skipkeys, ↩
    ensure_ascii, check_circular, allow_nan, cls, indent, separators, encoding, ↩
    default, sort_keys, **kw)
    241         cls is None and indent is None and separators is None and
    242         encoding == 'utf-8' and default is None and not sort_keys and not↩
         kw):
--> 243         return _default_encoder.encode(obj)
    244     if cls is None:
    245         cls = JSONEncoder

/opt/anaconda/lib/python2.7/json/encoder.pyc in encode(self, o)
    205         # exceptions aren't as detailed.  The list call should be roughly
    206         # equivalent to the PySequence_Fast that ''.join() would do.
```

```
--> 207         chunks = self.iterencode(o, _one_shot=True)
    208         if not isinstance(chunks, (list, tuple)):
    209             chunks = list(chunks)

/opt/anaconda/lib/python2.7/json/encoder.pyc in iterencode(self, o, _one_shot)
    268                 self.key_separator, self.item_separator, self.sort_keys,
    269                 self.skipkeys, _one_shot)
--> 270         return _iterencode(o, 0)
    271
    272 def _make_iterencode(markers, _default, _encoder, _indent, _floatstr,

/opt/anaconda/lib/python2.7/json/encoder.pyc in default(self, o)
    182
    183         """
--> 184         raise TypeError(repr(o) + " is not JSON serializable")
    185
    186     def encode(self, o):

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 as a string. 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)

>>> s = json.dumps(a, cls=CustomEncoder)
>>> s
'{"dtype": "set", "data": ["a", "c", "b", "e", "d", "g", "f"]}'
```

However, we want a Python set back when we decode. JSON will happily return our dictionary, but the data will be in a list. How do we tell it to convert our list back into a set? The answer is to build a custom decoder. Notice that we don't need to subclass anything.

```
accepted_dtypes = {'set': set}
def custom_decoder(dct):
    dt = accepted_dtypes.get(dct['dtype'], None)
    if dt is not None and 'data' in dct:
        return dt(dct['data'])
    return dct

>>> json.loads(s, object_hook=custom_decoder)
{u'a', u'b', u'c', u'd', u'e', u'f', u'g'}
```

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

However, datetime objects are not JSON serializable. Determine how best to serialize and deserialize a datetime object. The datetime object you serialize should be equal to the datetime object you get after deserializing.

Hint: You might want to read Problem 2 before designing your solution to this problem.

XML

XML is another data interchange format. It is a markup language rather than a object notation language. To understand XML, we need to understand what tags are. A tag is a special command enclosed in angled brackets (¡¿) that describe something about the data it encloses. For example, we can represent our car from above in the XML below.

```
<car>
    <make>Ford</make>
    <model>Focus</model>
    <year>2010</year>
    <color model='rgb'>255,30,30</color>
</car>
```

There are two strategies for reading XML data. We can read the data 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 is random access to any part of the document at any time. However, all of the XML must be loaded into memory to build this tree. Large XML files will consume huge amounts of memory if read as a tree.

To alleviate the burden of loading an entire XML document into memory all at once, we can read the file sequentially. When streaming the XML data, we are only reading a small chunk of the file at a time. There is no limit to size of XML document that we can process this way as memory usage will be constant. However, we sacrifice the random access that the tree gives us.

DOM

The DOM (Document Object Model) API allows you to work with an XML document as a tree. 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.

The motivation behind DOM is to represent an XML as a hierarchy of elements. This is accomplished by building a tree of the elements as the XML tags are read from the file. DOM is useful when we want random access to all of the XML document. This requires loading the entire file into memory. If you have a large XML file (a couple of megabytes)

DOM reads an entire XML into memory and builds a tree with the tag hierarchies on every parse. For large XML files, this could lead to massive memory consumption. The DOM tree of the car above would have `<car>` at the root element. This root element would have four children, `<make>`, `<model>`, `<year>`, and `<color>`. We would traverse this DOM tree just like we would any other tree structure. DOM trees can be searched by tag as well.

SAX

SAX, Simple API for XML, is essentially an XML state machine. This method of reading an XML file requires that you read the XML file as the parser would. It is a very fast, efficient way to read an XML file. The main advantage of this method for reading an XML file is memory conservation. A SAX parser reads XML sequentially instead of all at once. It doesn't need to load the entire file into memory.

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 into a single, high-level API for parsing and creating XML. 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.

```
<?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')

# manually traversing the tree
# we iterate through the element directly
# getchildren() is old and deprecated (not supported).
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
# searching for all tags equal to firstname
for n in root.iter('firstname'):
    print n.text

# we can also filter with multiple tags
# notice we use a set lookup in the conditional inside the generator expression
fields = {'firstname', 'lastname', 'phone'}
fi = (x for x in root.iter() if x.tag in fields)
for n in fi:
    print n.text

# we can even modify the document tree inplace
# let's remove Thor
# refer to the documentation of ElementTree for adding elements
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
for event, tag in f:
    print "{}: {}".format(tag.tag, tag.text)
    tag.clear()

# we can 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)
```

TCP/IP

The most common protocol that computers today use to communicate is TCP, Transmission Control Protocol. It is used in everything from checking email, uploading files, and browsing web pages. Being one of the core protocols of the internet protocol suite, it is often referred to as TCP/IP. There are four layers to the TCP/IP protocol.

1. Network Interface: This is the level of networking hardware such as routers and switches.
2. Internet: This level contains a group of protocols that handle routing and movement of data on a network.
3. Transport: The critical 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.
4. Application: Software that utilize the transport protocols to move information between computers. This layer includes protocols important for email, file transfers, and browsing the web.

TCP/IP has its origins in the mid 1970s. The TCP protocol dictates how computers connect to each, exchange bits of information called packets, and then close the connection. TCP/IP is very reliable, ordered, and error-checked.

Python has support for communicating via TCP in the standard library. A short demonstration will aide our discussion.

```
import socket

# define the parameters of the tcp connection
ip = '127.0.0.1' #the local machine
port = 33498 #arbitrary port number
size = 2048  # Normally 1024, but we want fast response

# create a new socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# and bind it to our desired address and port
s.bind((ip, port))
# start listening for requests
s.listen(1)

# Accept a request when one comes
conn, addr = s.accept()
print "Accepting connection from: ", addr
while True:
    # read 20 bytes from the incoming connection
    data = conn.recv(size)
    # if we have note received any data, we terminate the connection
    if not data:
```

```
23          break
    # otherwise we send back the data we received from the client
25      print "Echoing data: ", data
        conn.send(data)  # echo
27  conn.close()
```

tcp_server.py

```
1   #!/usr/bin/env python

3   import socket


5
    ip = '127.0.0.1'
7   port = 33498
    size = 2048
9   msg = "It's a beautiful networked world!"

11  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((ip, port))
13  s.send(msg)

15  data = s.recv(size)
    print "Received data: ", data
17  s.close()
```

tcp_client.py

We can start understanding the code above by seeing that a TCP connection is a link between two sockets. Those sockets can be on the same machine (as in the case above), or different machines. The machines are uniquely identified by an IP address. The IP address used in our example is a special IP address that always refers the local machine. Every computer has the ip

Let's start with defining some of the basic terms. Every computer on a network has an IP address. You can think of it as a mailing address for the computer. We communicate with a port on the computer via sockets on both the server and the client. This is analogous to a mailbox on the computer. There are 65535 available ports or mailboxes. Of those 65535 ports, about 250 are commonly used. In our script above, we use port 33498 to communicate. Ports 0 to 1023 are special reserved ports. For example, most web traffic flows through port 80 or 443. Email servers often uses ports 25, 110, 143, or 465. As you can see in the example code above, we create a socket on the server that we bind to the IP address and a port number. This socket will listen on that port for bits of network communication called packets.

On the client side, we create another socket that we bind to the same port. After the two sockets are defined, we can create a TCP connection. A TCP connection is a connection between two sockets. In our example, we have a client that initiates a connection and sends a message. The server sees an incoming message and then receives it. The message is sent in blocks, so we need to loop until we have received all of the blocks. As we receive each block, we send it right back to where it came from. The client receives these returned blocks and prints them to the console. We use a similar pattern to for every transfer of data over TCP. For a few connections, the amount of work the programmer has to do is not hard. However, imagine trying to request a complicated web page. We would have to manage possibly hundreds

of connections. We would naturally want to use a higher level protocol that takes care of the smaller details for us.

HTTP

HTTP stands for Hypertext Transfer Protocol. The protocol is centered around a request and response paradigm. A client makes a request to a server and the server replies with response. HTTP is an application layer networking protocol. This means it is a higher level protocol than TCP, taking care of many of the small details of TCP for us. It usually relies on the underlying TCP protocol to provide networking capabilities. There are several methods defined for HTTP, but the two most common are GET and POST. 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. We can send additional information with both GET and POST requests.

Every HTTP request consists of two parts: a header and a body. The headers contain important information about the request such as the type of request, encoding, among other things. We can add custom headers to any request to provide additional information. The body of the request contains the requested data. The body of a request 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

```
import httplib
conn = httplib.HTTPConnection("www.youtube.com")
conn.request("GET", "/")
resp = conn.getresponse()
if resp.status == 200:
    headers = resp.getheaders()
    data = resp.read()
conn.close()
print headers
print data      # very long string
```

We start by creating a connection to specific host. Then we make a request. In this case, we use GET request. The host we are connected to will respond and we retrieve the response. We will need to check the status of the response to know if our request was processed successfully. A status code of 200 means that everything went alright. We can now attempt to read the data of the response. At the end we explicitly close the connection.

This exchange is greatly simplified by the Requests library

```
import requests
r = requests.get("http://www.youtube.com")
r.close()
print r.headers
print r.content
```

Now, lets demonstrate various things we can do with HTTP requests. We will use a web service called HTTPBin which is very helpful in developing applications that make HTTP requests. When making a GET request, we can send along a list of parameters. These parameters should be a Python dictionary.

```
>>> data = {'key1': 0, 'key2': 1}
>>> r = requests.get("http://httpbin.org/get", params=data)
>>> print r.content
```

When we post to a server, we have the option of sending data. This data can be a file object, a dictionary or a string. To send our data via post, we first serialize it to JSON and then send the resulting string to the request.

```
>>> p = requests.post("http://httpbin.org/post", data=json.dumps(data))
>>> print p.content
```

Problem 2. You are assigned to implement a message board client. Your solution must be able to connect to a server with a given IP address and any specified port. Do not hard code the values in your client. You should prompt for them every time the script is run.

The first operation that your script will need to do is connect to the server and register the nickname that your user provides. Nicknames must be unique to a server, meaning no two users can have the same nickname.

The next task your solution should implement is an interface whereby a user can send and receive messages to and from the server. You may design your own user interface.

Your solution should be able to accept special commands that start with a double forward slash (//).

You client should accept a list of commands and perform the associated action.

Command	URL	Method	Action
//join			Join another channel
//nick	/changenick	POST	Change the user's nickname
//quit			Quit the message board
//pull	/message/pull	GET	Check for new messages and display them to the user
//push	/message/push	POST	Publish a new message on the message board

All requests to the server must be serialized JSON.

//push Push a new message to the server. Each message must have the fields: timestamp, content, channel, and nick. The timestamp will be used to identify when the message was sent. The content field will contain the body of the message. Channel will be the channel to

which the message is sent and the nick identifies who sent the message. Example:

```
{"content": "This is a test message.",
 "timestamp": "2014-07-09T15:15:44.331126",
 "channel": 1,
 "nick": "john"}
```

//pull Pull new messages from the server. Each request must have the fields: channel, timestamp, and nick. All messages posted after the timestamp will be retrieved. Channel tells the server which channel to fetch messages from. Nick identifies the user that is requesting the messages. Example:

```
{"timestamp": "2014-07-09T15:19:41.671257",
 "channel": 1,
 "nick": "john"}
```

//join Join a new channel. Typically, a user is only a member of one channel. Joining a new channel will change the channel they are currently listening to. However, you may design your client such that a user can subscribe to multiple channels. Only the client keep tracks of which channels the user is listening to.

//nick Change the nickname of the user. Message fields are: old_nick, new_nick. Server will return OK message if updating the new nickname was successful. Otherwise, the update will have failed, and nothing happened. Example:

```
{"old_nick": "john",
 "new_nick": "jack"}
```

Note that the server requires a timestamp for many of its transactions. The server will expect timestamps in ISO format. You will need to be able to send timestamps serialized into the ISO format. The server will return timestamps using ISO date formatting. You might need to be able to convert these timestamps into datetime objects. Dates in this format are of the form of `YYYY-mm-ddTHH:MM:SS.us`. You can parse this into a datetime object using the `strptime()` method of the datetime class. Note, `strptime()` does not natively accept microseconds, so you will have to process the string in parts.

Lab 9

MongoDB

Lab Objective: *In this lab we introduce MongoDB, a non-relational database system. MongoDB shares many similarities with JSON, and includes many of the same properties. We use MongoDB to investigate 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 impractical for some sets of data. Sometimes the relation model is too structured. Each item may not have the same attributes. For example, 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, hence the need for non-relational databases.

A new family of databases arose that attempted to solve the problem of non-relational attributes. Instead of designing a new relational database to meet every need, non-relational databases were created that can adapt the different items for specific scenarios. MongoDB is such a database. Several other databases, 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 has both a command line interface and Python bindings. This lab will use the official supported Python bindings, Pymongo. After being installed, Pymongo can be imported as with other standard libraries as follows:

```
from pymongo import MongoClient

# Create an instance of a client
# Connect on the default host 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. That is because it doesn't use relation tables. 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 meaningless attributes.

Creating and Removing Collections and Documents

A database stores collections, and a collection stores documents. This is the basic hierarchy of MongoDB. Each database can have several collections, each with its own documents. We need to create a database that will hold our collections. 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. 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.

```
f = list(col.find({'age': {'$lt': 24}, 'classes': {'$in': ['Art', 'English']}}))
```

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 9.1: MongoDB query operators

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.