Introduction to Programming

Joel Ross

March 25, 2018

2

Contents

About this Book

This book covers the fundamentals of computer programming as used for data science. It will introduce programming concepts such as syntax, data referencing, control structures, functions, data structures, abstraction, and debugging, as well as development tools and technologies (such as the command-line and version control). The goal is to develop skills in algorithmic thinking, abstraction, and debugging. It assumes no previous programming background. These materials were developed for the **INFX 511: Introduction to Programming for Data Science and Visualization** course taught at the University of Washington Information School; however they have been structured to be an online resource for anyone who wishes to learn programming.

Some chapters of this book have been developed in conjunction with Technical Foundations of Informatics, by Freeman and Ross.

This book is currently in **alpha** status. Visit us on GitHub to contribute improvements.

Chapter 1

Setting Up Your Machine

We'll be using a variety of different software programs to write, manage, and execute the code that we write. Unfortunately, one of the most frustrating and confusing barriers to working with code is simply getting your machine properly set up. This chapter aims to provide sufficient information for setting up your machine and troubleshooting the process.

Note that iSchool lab machines should have all appropriate software already installed and ready to use.

In short, you'll need to install the following programs: see below for more information / options.

- **Git**: A set of tools for tracking changes to computer code (especially when collaborating with others). This program is already installed on Macs.

    - **GitHub**: A web service for hosting code online. You don't actually need to *install* anything (GitHub uses `git`), but you'll need to sign up for the service.

- **Bash**: A *command-line interface* for controlling your computer. `git` is a command-line program so you'll need a command shell to use it. Macs already have a Bash program called *Terminal*. On Windows, installing `git` will also install a Bash shell called *Git Bash*, or you can try the Linux subsystem for Windows 10.

- **Visual Studio Code**: A lightweight text editor that supports programming in lots of different languages.

    - You are welcome to use another text editor if you wish; some further suggestions are included.

- **Python**: a general purpose programming language, but one that is often used when working with data. This will be the primary programming

11

language use in this book (though others will be discussed as well). "Installing Python" actually means installing tools that will let your computer understand and run Python code.

– **Anaconda**: is a *distribution* of the Python language, which includes a large number of additional add-on packages that we will be using (and can be difficult to install on their own). This is the recommended way of installing Python; we will be assuming that you have installed all of tools provided with the distribution.

The following sections have additional information about the purpose of each component, how to install it, and alternative configurations.

1.1   Git

`git` is a version control system that provides a set of commands that allow you to manage changes to written code, particularly when collaborating with other programmers (much more on this in Chapter 4). To start, you'll need to download and install the software. If you are on a Mac, `git` should already be installed.

If you are using a Windows machine, this will also install a program called *Git Bash*, which provides a text-based interface for executing commands on your computer. For alternative/additional Windows command-line tools, see below.

1.1.1   GitHub

GitHub is a website that is used to store copies of computer code that are being managed with `git` (think "Imgur for code"). Students in the INFX 511 course will use GitHub to turn in programming assignments.

In order to use GitHub, you'll need to create a free GitHub account, if you don't already have one. You should register a username that is identifiable as you (e.g., based on your name or your UW NetID). This will make it easier for others to determine who contributed what code, rather than needing to figure out who 'LeetDesigner2099' is. This can be the start of a professional account you may use for the rest of your career!

1.2   Command-line Tools (Bash)

The command-line provides a text-based interface for giving instructions to your computer (much more on this in Chapter 2). With this book, you'll use the command-line for navigating our computer's file structure, running programs,

and executing commands that allows you to keep track of changes to the code we write (i.e., version control with `git`).

In order to use the command-line, you will need to use a **command shell** (also called a *command prompt*). This is a program that provides the interface to type commands into. In particular, we'll be working with the Bash shell, which provides a particular set of commands common to Mac and Linux machines.

1.2.1   Command-line on a Mac

On a Mac you'll want to use the built-in app called *Terminal*. You can open Terminal by searching via Spotlight (hit Cmd (⌘) and Spacebar together, type in "terminal", then select the app to open it), or by finding it in the `Applications > Utilities` folder.

1.2.2   Command-line on Windows

On Windows, we recommend using **Git Bash**, which you should have installed along with `git` (above). Open this program to open the command-shell. This works great, since you'll primarily be using the command-line for performing version control.

- Note that Windows does come with its own command-prompt, called the *DOS Prompt*, but it has a different set of commands and features. *Powershell* is a more powerful version of the DOS prompt if you really want to get into the Windows Management Framework. But Bash is more common for the kinds of computer programming we'll be doing, and so we will be focusing on that set of commands.

Alternatively, the latest updates to Windows 10 (August 2016 or later) *does* includes a version of an integrated Bash shell. You can access this by enabling the subsystem for Linux and then running `bash` in the command prompt.

1.3   Text Editors

In order to produce computer code, you need somewhere to write it (and we don't want to write it in MS Word!) There are a variety of available programs that provide an interface for editing code. A major advantage of these programs is that they provide automatic formatting/coloring for easier interpretation of the code, along with cool features like auto-completion and integration with version control.

There are lots of different coding text editors out there, all of which have slightly different appearances and features. You only need to download and use one of

the following programs (we recommend **Visual Studio Code** as a default), but feel free to try out different ones to find something you like (and then evangelize about it to your friends!)

Programming involves working with many different file types, each detailed by their extension. It is useful to specify that your computer should show these extensions in the File Explorer or Finder; see instructions for Windows or for Mac to enable this.

1.3.1   Visual Studio Code

Visual Studio Code (or VS Code; not to be confused with Visual Studio) is a free, open-source editor developed by Microsoft. While it focuses on web programming and JavaScript, it readily supports lots of languages including Python, and provides a number of community-built extensions for adding even more features. Although fairly new, it is updated regularly and has has become one of my main editors for programming.

To install VS Code, follow the above link and Click the "Download" button to download the installer (e.g, `.exe`) file, then double-click on that to install the application.

Once you've installed VS Code, the trick to using it effectively is to get comfortable with the Command Palette. If you hit `Cmd+Shift+P`, VS Code will open a small window where you can search for whatever you want the editor to do. For example, if you type in `markdown` you can get list of commands related to Markdown files (including the ability to open up a preview).

- While VS Code can handle Python code just fine out of the box, to be most effective when writing code you'll want to add an extension with additional Python support. We recommend Don Jayamanne's extension: you can easily install this by using the command-palette (open it up and type `"extension"` to find the "Install Extensions" option). Note that you can disable the pervasive "style warnings" by adding `# pylint: skip-file` to the top of a script file.

For more information about using VS Code, see the documentation, which includes videos if you find them useful. There is also documentation for programming in Python specifically.

1.3.2   Atom

Atom is a text editor built by the folks at GitHub. It is very similar to VS Code in terms of features, but has a somewhat different interface and community. It has a similar command-palette to VS Code, and is arguably even nicer about

editing Markdown specifically (its built-in spell-check is a great feature, especially for documents that require lots of written text). This book was authored primarily in Atom.

1.3.3 Sublime Text

Sublime Text is a very popular text editor with excellent defaults and a variety of available extensions (though you'll need to manage and install extensions to achieve the functionality offered by other editors out of the box). While the software can be used for free, every 20 or so saves it will prompt you to purchase the full version.

1.3.4 PyCharm

**PyCharm(()) is a full-featured Python IDE (integrated development environment) developed by JetBrains. It has more features than you can shake a stick at, including a large number of features that won't be relevant for this book. It is more powerful, but also somewhat "heavier" (read: slower). You would be interested in the *Community Edition* of the software.

1.4 Python

The primary programming language you will use throughout the book is called **Python**. It's a very powerful, general-purpose programming language that is "friendly" enough that it is often used for information and data sciences. See Chapter 5 for a more in-depth introduction to the language,

Important note: There are two different versions of Python that exist in the world: Python 2 (latest: 2.7) and Python 3 (latest: 3.6). While mostly similar, the newer Python 3 version added a few basic differences that make it incompatible with Python 2. But a lot of existing programs were slow to change over to Python 3, effectively causing two different versions to exist in active use (kind of like people not switching from Windows XP). Python 3 is considered the "current" version, with Python 2 being "legacy"—and in fact is now "end of life" and will not be maintained.

In this book you will be working with **Python 3**. Thus you will want to make sure that you *install* Python 3, and that documentation and examples you reference are *for* Python 3 (rather than Python 2)!

In order to program with Python, you will need to install the ***Python Interpreter*** on your machine. This is a piece of software that is able to "read" code written in Python and use that code to control your computer, thereby "programming" it.

1.4.1   Anaconda

There are a number of ways to get Python installed on your machine: in fact, you may have a version (probably Python 2) installed on your computer already! However, the recommended approach for programming for data science is to install **Anaconda**. Anaconda is a pre-packaged bundle of common Python tools and packages that are commonly used in data science—many of which we will be utilizing (including Jupytr for running interactive "notebooks", and pandas for large-scale data work). And the best part: Anaconda makes it easy to get all the pieces installed and ready to use!

To install Anaconda (and Python along with it), simply download the **Python 3.6** installer, run the executable, and follow the instructions. This should install Python and all the required packages.

- You can confirm that everything is set up correctly by opening the *Command Shell* you installed above and typing `python --version`. You should see text that looks something like:

  ```
  Python 3.6.1 :: Anaconda 4.3.1 (x86_64)
  ```

  confirming that Python is installed and available.

Resources

Links to the recommended software are collected here for easy access:

- git (and Git Bash)
    - GitHub (sign up)
    - optional: Bash on Windows
- Visual Studio Code
- Python (Anaconda)

Chapter 2

The Command Line

The **command line** is an *interface* to a computer—a way for you (the human) to communicate with the machine. But unlike common graphical interfaces that use windows, icons, menus, and pointers, the command line is *text-based*: you type commands instead of clicking on icons. The command line lets you do everything you'd normally do by clicking with a mouse, but by typing in a manner similar to programming!

The command line is not as friendly or intuitive as a graphical interface: it's much harder to learn and figure out. However, it has the advantage of being both more powerful and more efficient in the hands of expert users. (It's faster to type than to move a mouse, and you can do *lots* of "clicks" with a single command). The command line is also used when working on remote servers or other computers that for some reason do not have a graphical interface enabled. Thus, command line is an essential tool for all professional developers, particularly when working with large amounts of data or files.

This chapter will give you a brief introduction to basic tasks using the command line: enough to get you comfortable navigating the interface and able to interpret commands.

2.1 Accessing the Command line

In order to use the command line, you will need to open a **command shell** (a.k.a. a *command prompt*). This is a program that provides the interface to type commands into. You should have installed a command shell (hereafter "the terminal") as part of setting up your machine.

Once you open up the shell (Terminal or Git Bash), you should see something like this (red notes are added):

Figure 2.1: An example of the command line in action (from Wikipedia).
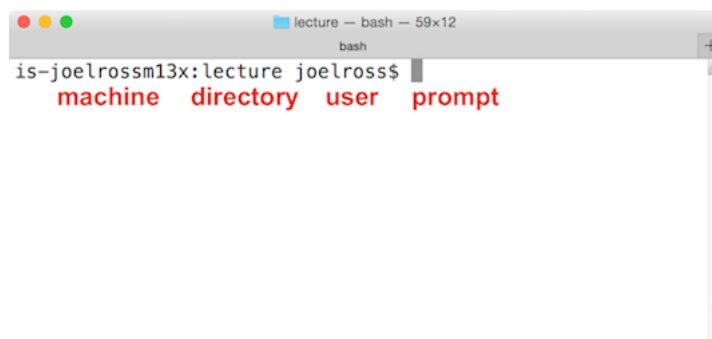
Figure 2.2: A newly opened command line.

This is the textual equivalent of having opened up Finder or File Explorer and having it show you the user's "Home" folder. The text shown lets you know:

- What **machine** you're currently interfacing with (you can use the command line to control different computers across a network or the internet).
- What **directory** (folder) you are currently looking at (~ is a shorthand for the "home directory").
- What **user** you are logged in as.

After that you'll see the **prompt** (typically denoted as the $ symbol), which is where you will type in your commands.

2.2 Navigating the Command Line

Although the command-prompt gives you the name of the folder you're in, you might like more detail about where that folder is. Time to send your first command! At the prompt, type:

```
pwd
```

This stands for **p**rint **w**orking **d**irectory (shell commands are highly abbreviated to make them faster to type), and will tell the computer to print the folder you are currently "in".

Fun fact: technically, this command usually starts a tiny program (app) that does exactly one thing: prints the working directory. When you run a command, you're actually executing a tiny program! And when you run programs (tiny or large) on the command line, it looks like you're typing in commands.

Folders on computers are stored in a hierarchy: each folder has more folders inside it, which have more folders inside them. This produces a tree structure which on a Mac may look like:

You describe what folder you are in putting a slash / between each folder in the tree: thus /Users/iguest means "the iguest folder, which is inside the Users folder".

At the very top (or bottom, depending on your point of view) is the **root** / directory-which has no name, and so is just indicated with that single slash. So /Users/iguest really means "the iguest folder, which is inside the Users folder, which is inside the *root* folder".

2.2.1 Changing Directories

What if you want to change folders? In a graphical system like Finder, you would just double-click on the folder to open it. But there's no clicking on the command line.
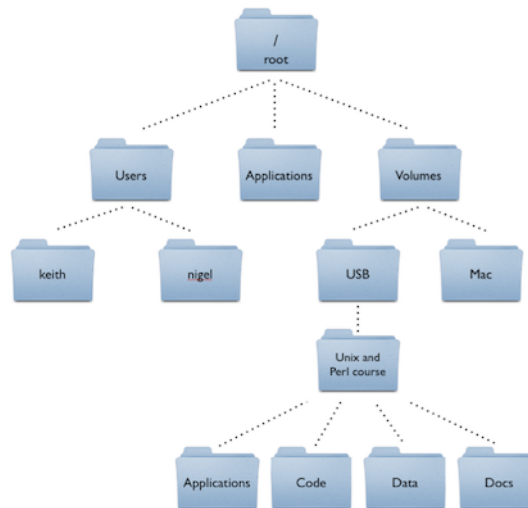
Figure 2.3: A Directory Tree, from Bradnam and Korf.

This includes clicking to move the cursor to an earlier part of the command you typed. You'll need to use the left and right arrow keys to move the cursor instead!

Protip: The up and down arrow keys will let you cycle though your previous commands so you don't need to re-type them!

Since you can't click on a folder, you'll need to use another command:

```
cd folder_name
```

The first word is the **command**, or what you want the computer to do. In this case, you're issuing the command that means **c**hange **d**irectory.

The second word is an example of an **argument**, which is a programming term that means "more details about what to do". In this case, you're providing a *required* argument of what folder you want to change to! (You'll of course need to replace `folder_name` with the name of the folder).

- Try changing to the `Desktop` folder, which should be inside the home folder you started in—you could see it in Finder or File Explorer!

- After you change folders, try printing your current location. Can you see that it has changed?

2.2.2 Listing Files

In a graphical system, once you've double-clicked on a folder, Finder will show you the contents of that folder. The command line doesn't do this automatically; instead you need another command:

```
ls [folder_name]
```

This command says to **lis**t the folder contents. Note that the *argument* here is written in brackets (`[]`) to indicate that it is *optional*. If you just issue the `ls` command without an argument, it will list the contents of the current folder. If you include the optional argument (leaving off the brackets), you can "peek" at the contents of a folder you are not currently in.

Warning: The command line can be not great about giving **feedback** for your actions. For example, if there are no files in the folder, then `ls` will simply show nothing, potentially looking like it "didn't work". Or when typing a **password**, the letters you type won't show (not even as `*`) as a security measure.

Just because you don't see any results from your command/typing, doesn't mean it didn't work! Trust in yourself, and use basic commands like `ls` and `pwd` to confirm any changes if you're unsure. Take it slow, one step at a time.

2.2.3 Paths

Note that both the **cd** and **ls** commands work even for folders that are not "immediately inside" the current directory! You can refer to *any* file or folder on the computer by specifying its **path**. A file's path is "how you get to that file": the list of folders you'd need to click through to get to the file, with each folder separated by a `/`:

```
cd /Users/iguest/Desktop/
```

This says to start at the root directory (that initial `/`), then go to `Users`, then go to `iguest`, then to `Desktop`.

Because this path starts with a specific directory (the root directory), it is referred to as an **absolute path**. No matter what folder you currently happen to be in, that path will refer to the correct file because it always starts on its journey from the root.

Contrast that with:

```
cd iguest/Desktop/
```

Because this path doesn't have the leading slash, it just says to "go to the `iguest/Desktop` folder *from the current location*". It is known as a **relative path**: it gives you directions to a file *relative to the current folder*. As such, the relative path `iguest/Desktop/` path will only refer to the correct location if

you happen to be in the `/Users` folder; if you start somewhere else, who knows where you'll end up!

You should **always** use relative paths, particularly when programming! Because you'll almost always be managing multiples files in a project, you should refer to the files *relatively* within your project. That way, you program can easily work across computers. For example, if your code refers to `/Users/your-user-name/project-name/data`, it can only run on the `your-user-name` account. However, if you use a *relative path* within your code (i.e., `project-name/data`), the program will run on multiple computers (crucial for collaborative projects).

You can refer to the "current folder" by using a single dot `.`. So the command

```
ls .
```

means "list the contents of the current folder" (the same thing you get if you leave off the argument).

If you want to go *up* a directory, you use *two* dots: `..` to refer to the **parent** folder (that is, the one that contains this one). So the command

```
ls ..
```

means "list the contents of the folder that contains the current folder".

Note that `.` and `..` act just like folder names, so you can include them anywhere in paths: `../../my_folder` says to go up two directories, and then into `my_folder`.

Protip: Most command shells like Terminal and Git Bash support **tab-completion**. If you type out just the first few letters of a file or folder name and then hit the `tab` key, it will automatically fill in the rest of the name! If the name is ambiguous (e.g., you type `Do` and there is both a `Documents` and a `Downloads` folder), you can hit `tab` *twice* to see the list of matching folders. Then add enough letters to distinguish them and tab to complete! This will make your life better.

Additionally, you can use a tilde `~` as shorthand for the home directory of the current user. Just like `.` refers to "current folder", `~` refers to the user's home directory (usually `/Users/USERNAME`). And of course, you can use the tilde as part of a path as well (e.g., `~/Desktop` is an *absolute path* to the desktop for the current user).

2.3   File Commands

Once you're comfortable navigating folders in the command line, you can start to use it to do all the same things you would do with Finder or File Explorer,

simply by using the correct command. Here is an short list of commands to get you started using the command prompt, though there are many more:

Command	Behavior
mkdir	**m**ake a **dir**ectory
rm	**rem**ove a file or folder
cp	**c**o**p**y a file from one location to another
open	opens a file or folder (Mac only)
start	opens a file or folder (Windows only)
cat	con**cat**enate (combine) file contents and display the results
history	show previous commands executed

Warning: The command line makes it **dangerously easy** to *permanently delete* multiple files or folders and *will not* ask you to confirm that you want to delete them (or move them to the "recycling bin"). Be very careful when using the terminal to manage your files, as it is very powerful.

Be aware that many of these commands **won't print anything** when you run them. This often means that they worked; they just did so quietly. If it *doesn't* work, you'll know because you'll see a message telling you so (and why, if you read the message). So just because you didn't get any output doesn't mean you did something wrong—you can use another command (such as **ls**) to confirm that the files or folders changed the way you wanted!

2.3.1   Learning New Commands

How can you figure out what kind of arguments these commands take? You can look it up! This information is available online, but many command shells (though *not* Git Bash, unfortunately) also include their own manual you can use to look up commands!

```
man mkdir
```

Will show the **man**ual for the **mkdir** program/command.

Because manuals are often long, they are opened up in a command line viewer called `less`. You can "scroll" up and down by using the arrow keys. Hit the `q` key to **q**uit and return to the command-prompt.

If you look under "Synopsis" you can see a summary of all the different arguments this command understands. A few notes about reading this syntax:

- Recall that anything in brackets [] is optional. Arguments that are not in brackets (e.g., `directory_name`) are required.

- **"Options"** (or "flags") for command line programs are often marked with a leading dash **-** to make them distinct from file or folder names. Options

```
● ● ●                     joelross — less • man mkdir — 95×33
                          ~ — less • man mkdir                              +

MKDIR(1)                     BSD General Commands Manual                    MKDIR(1)

NAME
     mkdir -- make directories

SYNOPSIS
     mkdir [-pv] [-m mode] directory_name ...

DESCRIPTION
     The mkdir utility creates the directories named as operands, in the order specified,
     using mode rwxrwxrwx (0777) as modified by the current umask(2).

     The options are as follows:

     -m mode
             Set the file permission bits of the final created directory to the specified
             mode.  The mode argument can be in any of the formats specified to the
             chmod(1) command.  If a symbolic mode is specified, the operation characters
             ``+'' and ``-'' are interpreted relative to an initial mode of ``a=rwx''.

     -p      Create intermediate directories as required.  If this option is not speci-
             fied, the full path prefix of each operand must already exist.  On the other
             hand, with this option specified, no error will be reported if a directory
             given as an operand already exists.  Intermediate directories are created
             with permission bits of rwxrwxrwx (0777) as modified by the current umask,
             plus write and search permission for the owner.

     -v      Be verbose when creating directories, listing them as they are created.

     The user must have write permission in the parent directory.

:▌
```

Figure 2.4: The `mkdir` man page.

may change the way a command line program behaves—like how you might set "easy" or "hard" mode in a game. You can either write out each option individually, or combine them: **`mkdir -p -v`** and **`mkdir -pv`** are equivalent.

- Some options may require an additional argument beyond just indicating a particular operation style. In this case, you can see that the `-m` option requires you to specify an additional `mode` parameter; see the details below for what this looks like.

- Underlined arguments are ones you choose: you don't actually type the word `directory_name`, but instead your own directory name! Contrast this with the options: if you want to use the `-p` option, you need to type `-p` exactly.

Command line manuals ("man pages") are often very difficult to read and understand: start by looking at just the required arguments (which are usually straightforward), and then search for and use a particular option if you're looking to change a command's behavior.

For practice, try to read the man page for `rm` and figure out how to delete a folder and not just a single file. Note that you'll want to be careful, as this is a good way to break things.

2.3.2 Wildcards

One last note about working with files. Since you'll often work with multiple files, command shells offer some shortcuts to talking about files with the same name. In particular, you can use an asterisk `*` as a **wildcard** when naming files. This symbol acts like a "wild" or "blank" tile in Scrabble–it can be "replaced" by any character (or any set of characters) when determining what file(s) you're talking about.

- `*.txt` refers to all files that have `.txt` at the end. **`cat *.txt`** would output the contents of every `.txt` file in the folder.

- `hello*` refers to all files whose names start with `hello`.

- `hello*.txt` refer to all files that start with `hello` and end with `.txt`, no matter how many characters are in the middle (including none!)

- `*.*` refers to all files that have an extension.

2.4 Dealing With Errors

Note that the syntax of these commands (how you write them out) is very important. Computers aren't good at figuring out what you meant if you aren't

really specific; forgetting a space may result in an entirely different action.

Try another command: **echo** lets you "echo" (print out) some text. Try echoing `"Hello World"` (which is the traditional first computer program):

```
echo "Hello world"
```

What happens if you forget the closing quote? You keep hitting "enter" but you just get that `>` over and over again! What's going on?

- Because you didn't "close" the quote, the shell thinks you are still typing the message you want to echo! When you hit "enter" it adds a *line break* instead of ending the command, and the `>` marks that you're still going. If you finally close the quote, you'll see your multi-line message printed!

IMPORTANT TIP If you ever get stuck in the command line, hit **ctrl-c** (The `control` and `c` keys together). This almost always means "cancel", and will "stop" whatever program or command is currently running in the shell so that you can try again. Just remember: "**ctrl-c** to flee".

(If that doesn't work, try hitting the `esc` key, or typing `exit`, `q`, or `quit`. Those commands will cover *most* command line programs).

Throughout this book, we'll discuss a variety of approaches to handling errors in computer programs. While it's tempting to disregard dense error messages, many programs do provide **error messages** that explain what went wrong. If you enter an unrecognized command, the terminal will inform you of your mistake:

```
lx
> -bash: lx: command not found
```

However, forgetting arguments yields different results. In some cases, there will be a default behavior (see what happens if you enter `cd` without any arguments). If more information is *required* to run a command, your terminal will provide you with a brief summary of the command's usage:

```
mkdir
> usage: mkdir [-pv] [-m mode] directory ...
```

Take the time to read the error message and think about what the problem might be before you try again.

2.5   Directing Output

So far all these commands have either modified the file system or printed some output to the terminal. But you can specify that you want the output to go somewhere else (e.g., to save it to a file for later). These are called **redi-**

rects. Redirect commands are usually single punctuation marks, because the commands want to be as quick to type (but hard to read!) as possible.

- `>` says "take output of command and put it in this file". For example `echo "Hello World" > hello.txt` will put the outputted text "Hello World" into a file called `hello.txt`. Note that this will replace any previous content in the file, or create the file if it doesn't exist. This is a great way to save the output of your command line work!

- `>>` says "take output of command and *append* it to the end of this file". This will keep you from overwriting previous content.

- `<` says "take input from this file". This is a much less common redirect.

- `|` says "take the output of this command and send it to the next command". For example, `cat hello.txt | less` would take the output of the `hello.txt` file and send it to the `less` program, which gives that arrow-based "scrolling" interface that man pages use.

Redirects are a more "advanced" usage of the command line, but now you know what those random symbols mean if you see them!

2.6 Shell Scripts

Shell commands are a way to tell the computer what to do—in effect, program it!

But often the instructions you want a computer to perform are more complex than a single command (even with redirects), or are something you want to be able to save and repeat later (beyond just looking it up in the `history`). It is useful if you can write down all the instructions in a single place, and then order the computer to *execute* all of those instructions at once. This list of instructions is called a **script**, with a list of shell commands called a **shell scripts**. Executing or "running" a script will cause each instruction (line of code) to be run *in order, one after the other*, just as if you had typed them in one by one. Writing scripts allows you to save, share, and re-use your work—by saving instructions in a file, you can easily check, change, and re-execute the list of instructions (assuming you haven't caused any irreversible side effects).

Bash shell scripts (also known as "Bash scripts") are generally written in files that end in the **.sh** extension (for **sh**ell). Once you've written a shell script, you can execute it on the command line simply by typing the file name as the command:

```
./my-script.sh
```

(The `./` indicates that we want to execute the file in the current folder, instead of looking for a program elsewhere on the computer as normally happens with

commands).

- **Important** On OS X or Linux you will need to give the file permission to be executed to run it. Use the `chmod` (**ch**ange **mod**e) command to add (`+`) **ex**ecution permissions to the file:

```
chmod +x my-script.sh
```

For *portability* (e.g., to work well across computers), the first line of any Bash shell script should be a a shebang that indicates which shell should be used:

```
#!/usr/bin/env bash
```

After that, you can begin listing the command your script will run, one after another (each on a new line):

```
#!/usr/bin/env bash

# anything after a '#' is a comment, and will not be executed
echo "Hello world"
echo "This is a second command"
```

- You can include blank lines between your command for readability, and add *comments* (notes to yourself that are not executed by the computer) by starting a line with `#`.

Shell scripts are an easy way to make "macros" or shortcuts for commands you want to run, and are common ways of sharing computer instructions with others.

Resources

- Learn Enough Command Line to be Dangerous
- Video series: Bash commands
- List of Common Commands (also here)

Chapter 3

Git and GitHub

A frightening number of people still email their code to each other, have dozens of versions of the same file, and lack any structured way of backing up their work for inevitable computer failures. This is both time consuming and error prone.

And that is why they should be using **git**.

This chapter will introduce you to `git` command-line program and the GitHub cloud storage service, two wonderful tools that track changes to your code (`git`) and facilitate collaboration (GitHub). Git and GitHub are the industry standards for the family of tasks known as **version control**. Being able to manage changes to your code and share it with others is one of the most important technical skills a programmer can learn, and is the focus of this (lengthy) chapter.

3.1   What is this *git* thing anyway?

THIS IS GIT. IT TRACKS COLLABORATIVE WORK
ON PROJECTS THROUGH A BEAUTIFUL
DISTRIBUTED GRAPH THEORY TREE MODEL.

COOL. HOW DO WE USE IT?

NO IDEA. JUST MEMORIZE THESE SHELL
COMMANDS AND TYPE THEM TO SYNC UP.
IF YOU GET ERRORS, SAVE YOUR WORK
ELSEWHERE, DELETE THE PROJECT,
AND DOWNLOAD A FRESH COPY.

Git is an example of a **version control system**. Eric Raymond defines version control as

> A version control system (VCS) is a tool for managing a collection
> of program code that provides you with three important capabilities:
> **reversibility**, **concurrency**, and **annotation**.

Version control systems work a lot like Dropbox or Google Docs: they allow multiple people to work on the same files at the same time, to view and "roll back" to previous versions. However, systems like git different from Dropbox in a couple of key ways:

1. New versions of your files must be explicitly "committed" when they are ready. Git doesn't save a new version every time you save a file to disk. That approach works fine for word-processing documents, but not for programming files. You typically need to write some code, save it, test it, debug, make some fixes, and test again before you're ready to save a new version.

2. For text files (which almost all programming files are), git tracks changes *line-by-line.* This means it can easily and automatically combine changes from multiple people, and gives you very precise information what what lines of code changes.

Like Dropbox and Google Docs, git can show you all previous versions of a file and can quickly rollback to one of those previous versions. This is often helpful in programming, especially if you embark on making a massive set of changes, only to discover part way through that those changes were a bad idea (we speak from experience here).

But where git really comes in handy is in team development. Almost all professional development work is done in teams, which involves multiple people working on the same set of files at the same time. Git helps the team coordinate all these changes, and provides a record so that anyone can see how a given file ended up the way it did.

There are a number of different version control systems in the world, but git is the de facto standard—particularly when used in combination with the cloud-based service GitHub.

3.1.1 Git Core Concepts

To understand how git works, you need to understand its core concepts. Read this section carefully, and come back to it if you forget what these terms mean.

- **repository (repo):** A database containing all the committed versions of all your files, along with some additional metadata, stored in a hidden subdirectory named `.git` within your project directory. If you want to sound cool and in-the-know, call a project folder a "repo."

- **commit:** A set of file versions that have been added to the repository (saved in the database), along with the name of the person who did the commit, a message describing the commit, and a timestamp. This extra tracking information allows you to see when, why, and by whom changes were made to a given file. Committing a set of changes creates a "snapshot"

of what that work looks like at the time—it's like saving the files, but more so.

- **remote:** A link to a copy of this same repository on a different machine. Typically this will be a central version of the repository that all local copies on your various development machines point to. You can push (upload) commits to, and pull (download) commits from, a remote repository to keep everything in sync.

- **merging:** Git supports having multiple different versions of your work that all live side by side (in what are called **branches**), whether those versions are created by one person or many collaborators. Git allows the commits saved in different versions of the code to be easily *merged* (combined) back together without you needing to manually copy and paste different pieces of the code. This makes it easy to separate and then recombine work from different developers.

3.1.2   Wait, but what is GitHub then?

Git was made to support completely decentralized development, where developers pull commits (sets of changes) from each other's machines directly. But most professional teams take the approach of creating one central repository on a server that all developers push to and pull from. This repository contains the authoritative version the source code, and all deployments to the "rest of the world" are done by downloading from this centralized repository.

Teams can setup their own servers to host these centralized repositories, but many choose to use a server maintained by someone else. The most popular of these in the open-source world is GitHub. In addition to hosting centralized repositories, GitHub also offers other team development features, such as issue tracking, wiki pages, and notifications. Public repositories on GitHub are free, but you have to pay for private ones.

In short: GitHub is a site that provides as a central authority (or clearing-house) for multiple people collaborating with git. Git is what you use to do version control; GitHub is one possible place where repositories of code can be stored.

3.2   Installation & Setup

This chapter will walk you through all the commands you'll need to do version control with git. It is written as a "tutorial" to help you practice what you're reading!

If you haven't yet, the first thing you'll need to do is install git. You should already have done this as part of setting up your machine.

The first time you use `git` on your machine, you'll need to configure the installation, telling git who you are so you can commit changes to a repository. You can do this by using the `git` command with the `config` option (i.e., running the `git config` command):

```
# enter your full name (without the dashes)
git config --global user.name "your-full-name"

# enter your email address (the one associated with your GitHub account)
git config --global user.email "your-email-address"
```
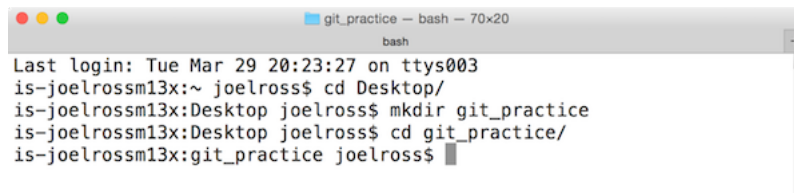
Setting up an SSH key for GitHub on your own machine is also a huge time saver. If you don't set up the key, you'll need to enter your GitHub password each time you want to push changes up to GitHub (which may be multiple times a day). Simply follow the instructions on this page to set up a key, and make sure to only do this on *your machine*.

3.2.1 Creating a Repo

The first thing you'll need in order to work with git is to create a **repository**. A repository acts as a "database" of changes that you make to files in a directory.

In order to have a repository, you'll need to have a directory of files. Create a new folder `git_practice` on your computer's Desktop. Since you'll be using the command-line for this course, you might as well practice creating a new directory programmatically:

Figure 3.1: Making a folder with the command-line.

You can turn this directory *into* a repository by telling the `git` program to run the `init` action:

```
# run IN the directory of project
# you can easily check this with the "pwd" command
git init
```

This creates a new *hidden* folder called `.git` inside of the current directory (it's hidden so you won't see it in Finder, but if you use `ls -a` (list with the **all** option) you can see it there). This folder is the "database" of changes that you will make—git will store all changes you commit in this folder. The presence

of the `.git` folder causes that directory to become a repository; we refer to the whole directory as the "repo" (an example of synechoche).

- Note that because a repo is a single folder, you can have lots of different repos on your machine. Just make sure that they are in separate folders; folders that are *inside* a repo are considered part of that repo, and trying to treat them as a separate repository causes unpleasantness. **Do not put one repo inside of another!**

3.2.2   Checking Status

Now that you have a repo, the next thing you should do is check its **status**:

```
git status
```

The `git status` command will give you information about the current "state" of the repo. For example, running this command tells us a few things:

- That you're actually in a repo (otherwise you'll get an error)
- That you're on the `master` branch (think: line of development)
- That you're at the initial commit (you haven't committed anything yet)
- That currently there are no changes to files that you need to commit (save) to the database
- *What to do next!*

That last point is important. Git status messages are verbose and somewhat awkward to read (this is the command-line after all), but if you look at them carefully they will almost always tell you what command to use next.

If you are ever stuck, use `git status` to figure out what to do next!

This makes `git status` the most useful command in the entire process. Learn it, use it, love it.

3.3   Making Changes

Since `git status` told you to create a file, go ahead and do that. Using your favorite editor, create a new file `books.md` inside the repo directory. This Markdown file should contain a *list* of 3 of your favorite books. Make sure you save the changes to your file to disk (to your computer's harddrive)!

3.3.1   Adding Files

Run `git status` again. You should see that git now gives a list of changed and "untracked" files, as well as instructions about what to do next in order to save those changes to the repo's database.

The first thing you need to do is to save those changes to the **staging area**. This is like a shopping cart in an online store: you put changes in temporary storage before you commit to recording them in the database (e.g., before hitting "purchase").

We add files to the staging area using the `git add` command:

```
git add filename
```

(Replacing `filename` with the name/path of the file/folder you want to add). This will add a single file *in its current saved state* to the staging area. If you change the file later, you will need to re-add the updated version.

You can also add all the contents of the directory (tracked or untracked) to the staging area with:

```
git add .
```

(This is what I tend to use, unless I explicitly don't want to save changes to some files.)

Add the `books.md` file to the staging area. And of course, now that you've changed the repo (you put something in the staging area), you should run `git status` to see what it says to do. Notice that it tells you what files are in the staging area, as well as the command to *unstage* those files (remove them from the "cart").

3.3.2 Committing

When you're happy with the contents of your staging area (e.g., you're ready to purchase), it's time to **commit** those changes, saving that snapshot of the files in the repository database. We do this with the `git commit` command:

```
git commit -m "your message here"
```

The `"your message here"` should be replaced with a short message saying what changes that commit makes to the repo (see below for details).

WARNING: If you forget the `-m` option, git will put you into a command-line *text editor* so that you can compose a message (then save and exit to finish the commit). If you haven't done any other configuration, you might be dropped into the ***vim*** editor. Type **:q** (**colon** then **q**) and hit enter to flee from this horrid place and try again, remembering the `-m` option! Don't panic: getting stuck in *vim* happens to everyone.

3.3.2.1   Commit Message Etiquette

Your commit messages should be informative about what changes the commit is making to the repo. `"stuff"` is not a good commit message. `"Fix critical authorization error"` is a good commit message.

Commit messages should use the **imperative mood** (`"Add feature"` not `"added feature"`). They should complete the sentence:

> If applied, this commit will **{your message}**

Other advice suggests that you limit your message to 50 characters (like an email subject line), at least for the first line—this helps for going back and looking at previous commits. If you want to include more detail, do so after a blank line.

A specific commit message format may also be required by your company or project team. See this post for further consideration of good commit messages.

Finally, be sure to be professional in your commit messages. They will be read by your professors, bosses, coworkers, and other developers on the internet. Don't join this group.

After you've committed your changes, be sure and check `git status`, which should now say that there is nothing to commit!

3.3.3   Commit History

You can also view the history of commits you've made:

```
git log [--oneline]
```

This will give you a list of the *sequence* of commits you've made: you can see who made what changes and when. (The term **HEAD** refers to the most recent commit). The optional `--oneline` option gives you a nice compact version. Note that each commit is listed with its SHA-1 hash (the random numbers and letters), which you can use to identify each commit.

3.3.4   Reviewing the Process

This cycle of "edit files", "add files", "commit changes" is the standard "development loop" when working with git.

In general, you'll make lots of changes to your code (editing lots of files, running and testing your code, etc). Then once you're at a good "break point"—you've got a feature working, you're stuck and need some coffee, you're about to embark on some radical changes—you will add and commit your changes to make sure you don't lose any work and you can always get back to that point.

3.3.4.1 Practice

For further practice using git, perform the following steps:

1. **Edit** your list of books to include two more books (top 5 list!)
2. **Add** the changes to the staging area
3. **Commit** the changes to the repository

Be sure and check the status at each step to make sure everything works!

3.3.5 The `.gitignore` File

Sometimes you want git to always ignore particular directories or files in your project. For example, if you use a Mac and you tend to organize your files in the Finder, the operating system will create a hidden file in that folder named `.DS_Store` (the leading dot makes it "hidden") to track the positions of icons, which folders have been "expanded", etc. This file will likely be different from machine to machine. If it is added to your repository and you work from multiple machines (or as part of a team), it could lead to a lot of merge conflicts (not to mention cluttering up the folders for Windows users).

You can tell git to ignore files like these by creating a special *hidden* file in your project directory called `.gitignore` (note the leading dot). This file contains a *list* of files or folders that git should "ignore" and pretend don't exist. The file uses a very simple format: each line contains the path to a directory or file to ignore; multiple files are placed on multiple lines. For example:

```
# This is an example .gitignore file

# Mac system file; the leading # marks a comment
.DS_Store

# example: don't check in passwords or ssl keys!
secret/my_password.txt

# example: don't include large files or libraries
movies/my_four_hour_epic.mov
```

Note that the easiest way to create the `.gitignore` file is to use your preferred text editor (e.g., VS Code); select `File > New` from the menu and choose to make the `.gitignore` file *directly inside* your repo.

If you are on a Mac, we **strongly suggest** *globally ignoring* your `.DS_Store` file. There's no need to ever share or track this file. To always ignore this file on your machine, you can create a "global" `.gitignore` file (e.g., in your ~ home directory), and then tell `git` to always exclude files listed there through the `core.excludesfile` configuration option:

```
# append `.DS_Store` in central file
echo .DS_Store >> ~/.gitignore

# always ignore files listed in that central file
git config --global core.excludesfile ~/.gitignore
```

See this article for more information.

3.4   GitHub and Remotes

Now that you've gotten the hang of git, let's talk about GitHub. GitHub is an online service that stores copies of repositories in the cloud. These repositories can be *linked* to your **local** repositories (the one on your machine, like you've been working with so far) so that you can synchronize changes between them.

- The relationship between git and GitHub is the same as that between your camera and Imgur: **git** is the program we use to create and manage repositories; GitHub is simply a website that stores these repositories. So we use git, but upload to/download from GitHub.

Repositories stored on GitHub are examples of **remotes**: other repos that are linked to your local one. Each repo can have multiple remotes, and you can synchronize commits between them.

Each remote has a URL associated with it (where on the internet the remote copy of the repo can be found), but they are given "alias" names (like browser bookmarks). By convention, the remote repo stored on GitHub's servers is named **origin**, since it tends to be the "origin" of any code you've started working on.

Remotes don't need to be stored on GitHub's computers, but it's one of the most popular places to put repos.

3.4.1   Forking and Cloning

In order to use GitHub, you'll need to **create a free GitHub account**, which you should have done as part of setting up your machine.

Next, you'll need to download a copy of a repo from GitHub onto your own machine. **Never make changes or commit directly to GitHub**: all development work is done locally, and changes you make are then uploaded and *merged* into the remote.

Start by visiting this link. This is the web portal for an existing repository. You can see that it contains one file (README.md, a Markdown file with a description

of the repo) and a folder containing a second file. You can click on the files and folder to view their source online, but again you won't change them there!

Just like with Imgur or Flickr or other image-hosting sites, each GitHub user has their own account under which repos are stored. The repo linked above is under the course book account (`infx511`). And because it's under our user account, you won't be able to modify it—just like you can't change someone else's picture on Imgur. So the first thing you'll need to do is copy the repo over to *your **own** account on GitHub's servers*. This process is called **forking** the repo (you're creating a "fork" in the development, splitting off to your own version).

- To fork a repo, click the **"Fork"** button in the upper-right of the screen:
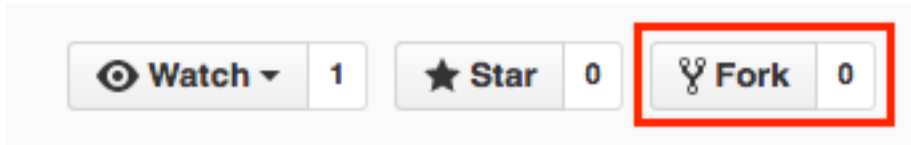
Figure 3.2: The fork button on GitHub's web portal.

   This will copy the repo over to your own account, so that you can upload and download changes to it!

   Students in the INFX 511 course will be forking repos for class and lab exercises, but *not* for homework assignments (see below)

Now that you have a copy of the repo under your own account, you need to download it to your machine. We do this by using the `clone` command:

```
git clone [url]
```

This command will create a new repo (directory) *in the current folder*, and download a copy of the code and all the commits from the URL you specify.

- You can get the URL from the address bar of your browser, or you can click the green "Clone or Download" button to get a popup with the URL. The little icon will copy the URL to your clipboard. **Do not** click "Open in Desktop" or "Download Zip".

- Make sure you clone from the *forked* version (the one under your account!)

   Warning also be sure to `cd` out of the `git_practice` directory; you don't want to `clone` into a folder that is already a repo; you're effectively creating a *new* repository on your machine here!

Note that you'll only need to `clone` once per machine; `clone` is like `init` for repos that are on GitHub—in fact, the `clone` command *includes* the `init` command (so you do not need to init a cloned repo).

3.4.2   Pushing and Pulling

Now that you have a copy of the repo code, make some changes to it! Edit the `README.md` file to include your name, then `add` the change to the staging area and `commit` the changes to the repo (don't forget the `-m` message!).

Although you've made the changes locally, you have not uploaded them to GitHub yet—if you refresh the web portal page (make sure you're looking at the one under your account), you shouldn't see your changes yet.

In order to get the changes to GitHub, you'll need to `push` (upload) them to GitHub's computers. You can do this with the following command:

```
git push origin master
```

This will push the current code to the `origin` remote (specifically to its `master` branch of development).

- When you cloned the repo, it came with an `origin` "bookmark" to the original repo's location on GitHub!

Once you've **pushed** your code, you should be able to refresh the GitHub webpage and see your changes to the README!

If you want to download the changes (commits) that someone else made, you can do that using the `pull` command, which will download the changes from GitHub and *merge* them into the code on your local machine:

```
git pull
```

Because you're merging as part of a `pull`, you'll need to keep an eye out for **merge conflicts**! These will be discussed in more detail in chapter 14.

Pro Tip: always `pull` before you `push`. Technically using `git push` causes a merge to occur on GitHub's servers, but GitHub won't let you push if that merge might potentially cause a conflict. If you `pull` first, you can make sure your local version is up to date so that no conflicts will occur when you upload.

3.4.3   Reviewing The Process

Overall, the process of using git and GitHub together looks as follows:

3.4.4   Course Assignments on GitHub

For students in INFX 511: While class and lab work will use the "fork and clone" workflow described above, homework assignments will work slightly differently. Assignments in this course are configured using GitHub Classroom, which provides each student *private* repo (under the class account) for the assignment.

Each assignment description in Canvas contains a link to create an assignment repo: click the link and then **accept the assignment** in order to create your own code repo. Once the repository is created, you should `clone` it to your local machine to work. **Do not fork your assignment repo**.

DO NOT FORK YOUR ASSIGNMENT REPO.

After `cloning` the assignment repo, you can begin working following the workflow described above:

1. Make changes to your files
2. **Add** files with changes to the staging area (`git add .`)
3. **Commit** these changes to take a repo (`git commit -m "commit message"`)
4. **Push** changes back to GitHub (`git push origin master`) to turn in your work.

Repeat these steps each time you reach a "checkpoint" in your work to save it both locally and in the cloud (in case of computer problems).

3.5 Command Summary

Whew! You made it through! This chapter has a lot to take in, but really you just need to understand and use the following half-dozen commands:

- `git status` Check the status of a repo
- `git add` Add file to the staging area
- `git commit -m "message"` Commit changes
- `git clone` Copy repo to local machine
- `git push origin master` Upload commits to GitHub
- `git pull` Download commits from GitHub

Using git and GitHub can be challenging, and you'll inevitably run into issues. While it's tempting to ignore version control systems, **they will save you time** in the long-run. For now, do your best to follow these processes, and read any error messages carefully. If you run into trouble, try to understand the issue (Google/StackOverflow), and don't hesitate to ask for help.

Resources

- Git and GitHub in Plain English
- Atlassian Git Tutorial
- Try Git (interactive tutorial)
- GitHub Setup and Instructions
- Official Git Documentation

- Git Cheat Sheet

Chapter 4

Introduction to Python

Python is an extremely powerful open-source *programming language*. It is considered a "high-level", "general purpose" language in that offers strong abstractions on computer instructions (and in fact often reads like badly-punctuated English), and can be used effectively for a wide variety of purposes. Python is one of the most popular programming languages in the world, and very approachable for beginners and experts alike. Python will be the primary programming language for this course.

4.1 Programming with Python

Python is a **high-level, general-purpose programming language** that can be used to declare complex instructions for computers (e.g., for working with information and data). It is an **open-source** programming language, which means that it is free and continually improved upon by the Python community.

- *Fun Fact:* Python is named after British comedy troupe "Monty Python", because it seemed like a good idea at the time.

So far you've leveraged formal language to give instructions to your computers, such as by writing syntactically-precise instructions at the command line. Programming in Python will work in a similar manner: you will write instructions using Python's special language and syntax, which the computer will **interpret** as instructions for how to work with data.

Indeed, Python is an **interpreted language**, in that the computer (specifically the *Python Interpreter*) translates the high-level language into machine language *on the fly at the time you run it ("at runtime")*. The interpreter will read and execute one line of code at a time, executing that line before it even begins to consider the next. This is in contrast with *compiled languages* (like C or Java)

that has the computer do the translation in one pass, and then only execute the program after the whole thing is converted to machine language.

- This means that Python is technically a little slower to execute commands than C or Java because it needs to both translate and execute at once, but not enough that we'll notice.

While it's possible to execute Python instructions one at a time, it's much more common to write down all of the instruction in a single **script** so that the computer can execute them all at once. Python scripts are saved as files with the `.py` extension, You can author Python scripts in any text editor (such as VS Code), but we'll also write and run Python code in an interactive web application called Jupyter.

4.1.1   Versions

Python was first published by Dutch programmer Guido Van Rossum in 1991. Although the language is open-source (and so can be contributed to by anyone), Van Rossum still retains a lot of influence over where the language goes, and is known as Python's "Benevolent Dictator for Life".

Version 2.0 of Python was released in 2000, and version 3.0 was released in 2008. However, unlike with many other programming languages that release new versions, Python 2 and Python 3 are **not** compatible with one another: Python 2 code usually won't execute with Python 3 interpreter, and Python 3 code usually won't execute with a Python 2 interpreter.

Python 3 involved a major restructuring which included a number of "breaking" changes to core parts of the language. The most noticeable of these are that all text strings support Unicode (non-English characters), `print` is a function not a statement, and integers don't use floor devision (these concepts are discussed in more detail below).

In short, Python 3 is considered the "current" version, and Python 2 is considered the "legacy" version (and will not be supported beyond). However some external libraries refused or failed to update to Python 3. Thus in practice there are two incompatible versions of Python that are used in the world. See Python2orPython3 for more details.

- In this book, we will be using **Python 3**, particularly as it fixes some issues that often trip up beginners, is supported by all major libraries in a more effective way, and will still be supported in the near future!

4.2 Running Python Scripts

Python scripts (programs) are just a sequence of instructions, and there are a couple of different ways in which we can tell the computer to execute these instructions.

4.2.1 On the Command Line

It is possible to issue Python instructions (run lines of code; called **statements**) one-by-one at the command-line by starting an **interactive Python session** within your terminal. This will allow you to type Python code directly into the terminal, and your computer will interpret and execute each line of code (if you just typed Python syntax directly into the terminal, your computer wouldn't understand it).

With Python installed, you can start an interactive Python session on a by running the **python** program (simply type the `python` command if you've installed it on the `PATH` via Anaconda).

- On Windows using the Git Bash shell, you may need to utilize winpty to connect the Bash shell to the Python output: see here for an example). On my machine, using `winpty python` works well.

Note that if you haven't installed Python via Anaconda, or have multiple versions installed, you may need to use the `python3` command to run a Python 3 session.

On the iSchool lab machines, you can switch from Python 2 to Python 3 using the command:

```
source activate python3
```

Which will "activate" the Python 3 environment installed by Anaconda.
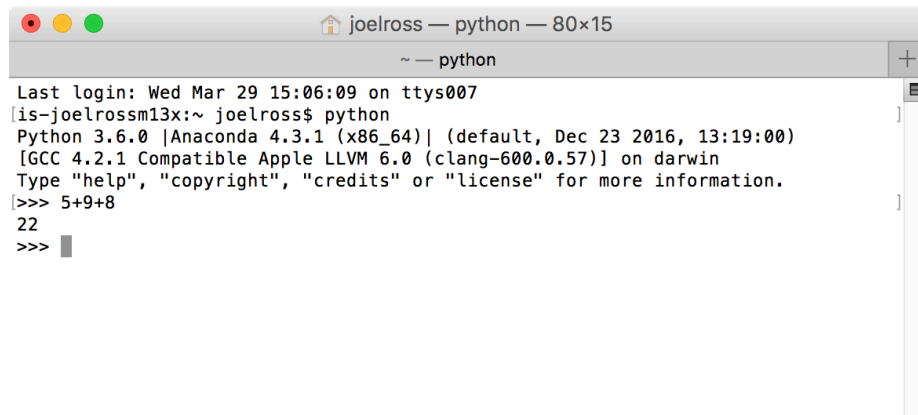
The `python` command will start the interactive session, providing some information about the version of Python being run:

Once you've started running an interactive Python session, you can begin entering one line of code at a time at the prompt (>>>). This is a nice way to experiment with the Python language or to quickly run and test some code.

- You can exit this session by typing the `quit()` command, or hitting `ctrl-z` (followed by Enter on Windows).

It is more common though to run entire scripts (`.py` files) from the command-line by using the `python` command and specifying the script file you wish to execute:
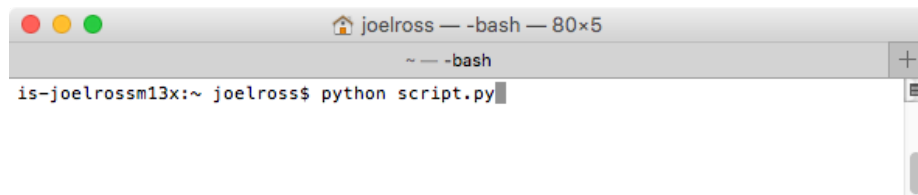
This will cause each line in the script to be run one at a time. This is the "normal" way of running Python programs: you write Python scripts using an

Figure 4.1: Interactive Python session

Figure 4.2: Run Python script from the command line

editor such as VS Code, and then execute those scripts on the command-line to see them in action.

4.2.2 Jupyter Notebooks

Another popular and effective way of writing and running Python code is to utilize a Jupyter Notebook. A Jupyter Notebook is a **web appplications** (that runs in your web browser) and provides an interactive platform for running Python code: you are able to write and execute Python code write in the browser, similar to an interactive session (but with the ability to write multiple lines of code).

- Jupter is a portmanteau of "Julia", "Python" and "R", which are the programming languages the application was designed to support. Each notebook runs on an individual **kernel** or language interpreter; you'll want to be sure you're running a Python 3 kernel for this course.

Code is authored in individual "notebooks", which are collections of *cells* (blocks of code or text). Cells can also include Markdown content, allowing notebooks to act as interactive documents with embedded, runnable code.

The Jupyter program is part of the Anaconda package, and so should be installed along with Python. You can start Jupyter from the command line with:

```
jupyter notebook
```

This will open up your web browser with an interface to create or open a notebook relative to the folder you started Jupyter from:

You can also open a particular notebook (which are saved as files with the `.ipynb` extension) by passing the notebook name as an argument:

```
jupyter notebook my-notebook.ipynb
```

- To stop running the Jupyter notebook server, hit `ctrl-c` in the terminal and follow the prompts.

Jupyter notebooks are made up of a series of *cells*, each representing a set of code or text. Cells by default are used for Python code, but can also be set to use Markdown by selecting from the dropdown menu.

You can type code directly into the cells, just as if you were typing into a text editor. In order to execute the code in a cell, press **shift-enter** or select `Cell > Run Cell` from the menu. Any output from the cell will appear immediately below it. You can add new cells by hitting the "plus" button on the toolbar.

- **Pro-tip**: There are lots of keyboard shortcuts available for working with cells. Click the keyboard icon on the toolbar to see a list of shortcuts.

4.3   Comments

Before discussing how to program with Python, we need to talk about a piece of syntax that lets you comment your code. In programming, **comments** are bits of text that are *not interpreted as computer instructions*—they aren't code, they're just notes about the code! Since computer code can be opaque and difficult to understand, we use comments to help write down the meaning and *purpose* of our code. While a computer is able to understand the code, comments are there to help *people* understand. This is particularly imporant when someone else will be looking at your work—whether that person is a collaborator, or is simply a future version of you (e.g., when you need to come back and fix something and so need to remember what you were even thinking). Comments should be clear, concise, and helpful—they should provide information that is not "obvious" from the code itself.

In Python, we mark text as a comment by putting it after the pound/hashtag symbol (`#`). Everything from the `#` until the end of the line is a comment. We put descriptive comments *immediately above* the code it describes, but you can also put very short notes at the end of the line of code (preferably following two spaces):

```python
# Set how many bottles of beer are on the wall
bottles = 99 - 1  # 98
```

(You may recognize this `#` syntax and commenting behavior from the command-line and git modules. That's because the same syntax is used in a Bash shell!)

4.4   Variables

Since computer programs involve working with lots of *information*, we need a way to store and refer to this information. We do this with **variables**. Variables are labels for information; in Python, you can think of them as "nametags" for data. After giving data a variable nametag, you can then refer to that data by its name.

Variable names can contain any combination of letters, numbers, or underscores (`_`). Variables names must begin with a letter. Note that like everything in programming, variable names are case sensitive. It is best practice to make variable names descriptive and information about what data they contain. `a` is not a good variable name. `cups_of_coffee` is a good variable name. To comply with Google's Style Guidelines variables should be in "snake_case": **all lower-case letters, separated by underscores (`_`)**.

We call putting information in a variable **assigning** that value to the variable. We do this using the *assignment operator* `=`. For example:

```
# Stores the number 7 into a variable called shoe.size
shoe_size = 7
```

- *Notice:* the variable name goes on the left, the value goes on the right!
- It is good style to put white spaces around the = operator to keep it easy to read.

You can see what value (data) is inside a variable by either typing that variable name as a line of code, or by using Python's built-in `print()` function (more on functions later):

```
print(shoe_size)
## 7
```

You can also use **mathematical operators** (e.g., `+`, `-`, `/`, `*`) when assigning values to variables. For example, you could create a variable that is the sum of two numbers as follows:

```
x = 3 + 4
```

- A combination of variables, values, operators, or functions that the interpreter needs to *evaluate* is called an **expression**.

Once a value (like a number) is assigned a variable—it is *in* a variable—you can use that variable in place of any other value. So all of the following are valid:

```
x = 2  # store 2 in x
y = 9  # store 9 in y
z = x + y  # store sum of x and y in z
print(z)  # 11
z = z + 1  # take z, add 1, and store result back in z
print(z)  # 12
```

Important despite using an equal sign (=), variable assignments *do not* specify equalities! Assigning one variable to another does not cause them to be "linked"; changing one variable will leave the other unchanged:

```
x = 3
y = 4
x = y  # assign the value of y (4) to x
print(x)  # 4
y = 5
print(x)  # 4;
```

Changing the value named y does not change the value named y, even if the value from y was put into the x variable at some point.

- *Tip*: when reading or writing code, try to think "x is assigned 3" or "x gets the value 3", rather than "x equals 3".

4.4.1   Data Types

In the example above, we stored **numeric** values in variables. Python is a **dynamically typed language**, which means that we *do not* need to explicitly state what type of information will be stored in each variable we create. The Python interpreter is intelligent enough to understand that if we have code `x = 7`, then `x` will contain a numeric value (and so we can do math upon it!)

You can "look up" the type of a particular value by using the `type()` function:

```python
type(7)  # integer
## <class 'int'>

type("Hello")  # string
## <class 'str'>
```

- Type is also referred to as `class` (i.e., "classification"). **Classes** are ways of structuring information and behavior in computer programs; we will discuss them in more detail later.

There are a number of "basic types" for data in Python. The two most common are:

- **Numeric Types**: Python has two data types used to represent numbers: `int` (integer) for *whole numbers* like `7` and `float` for decimals like `3.14` (because the decimal is "floating" in the middle).

  We can use **mathematical operators** on numeric data (such as `+`, `-`, `*`, `/`, etc.). There are also numerous functions that work on numeric data (such as for calculating sums or averages). You can mix and match ints and floats in mathematical expressions; the result will be a float if either of the operands are floats. Division (`/`) always produces a float; use the "integer division operator" `//` to force the result to be a whole number (rounded down).

- **Strings**: Python uses the `str` (string) type to store textual data. `str` values contain *strings* of characters, which are the things you type with a keyboard (including digits, spaces, punctuation, and even line breaks). You specify that some characters are a string by surrounding them in either single quotes (`'`) or double quotes (`"`).

  ```python
  # Create a string variable `famous_poet` with the value "Bill Shakespeare"
  famous_poet = "Bill Shakespeare"
  ```

  Note that string literals (the text in quotes) are *values* just like numeric literals (numbers), so can be assigned to variables!

  - Using "triple-quotes" will allow you to specify a multi-line string:

    ```python
    message = """Dear Professor,
                  My dog ate my homework.
    ```

```
                    Thank you,
                        A. Student
            """
```

Strings support the **+** operator, which performs **concatenation** (combining two strings together). For example:

```
# concatenate 3 strings. The middle string is a space character
full_name = "Bill" + " " + "Shakespeare"
```

The operands must both be strings. Also notice that concatenating strings doesn't add any additional characters such as spaces between words; you need to handle such punctuation yourself!

There are many built-in functions for working with strings, and in fact strings support a variety of functions you can call *on* them. See Chapter 6 for details.

There are many other data types as well; more will be introduced throughout the remaining modules.

Helpful tip: because variables can any type of data, it is often useful to name variables based on their data type. This helps keep the purpose of code clear in your mind, and makes it easier to read and understand what is happening:

```
password_num = 12345  # variable name indicates a number, so can do math on it
password_str = "12345"  # variable name indicates a string, so can print it
```

It is possible to convert from one type to another by using a **type converter function**, which is usually named after the type you wish to convert *to*:

```
int(3.14)    # 3
int(5.98)    # 5 (takes the floor value)
float(6)     # 6.0
str(2)       # "2"
int("2")     # 2
int("two")   # ValueError!
```

This is particularly useful when you wish to include a number in a string:

```
favorite_number = 12
message = "My favorite number is "+str(favorite_number)
print(message)  # My favorite number is 12
```

4.5 Getting Help

As with any programming language, when working in Python you will inevitably run into problems, confusing situations, or just general questions. Here are a

few ways to start getting help.

1. **Read the error messages**: If there is an issue with the way you have written or executed your code, Python will often print out an error message. Do you best to decipher the message (read it carefully, and think about what is meant by each word in the message), or you can put it directly into Google to get more information. You'll soon get the hang of interpreting these messages (and how to resolve common ones) if you don't panic when one appears. *Remember: making mistakes is normal!*

2. **Google**: When you're trying to figure out how to do something, it should be no surprise that Google is often the best resource. Try searching for queries like `"how to <DO THING> in Python"`. More frequently than not, your question will lead you to a Q/A forum called StackOverflow (see below), which is a great place to find potential answers.

3. **StackOverflow**: StackOverflow is an amazing Q/A forum for asking/answering programming questions. Most basic questions have already been asked/answered here. However, don't hesitate to post your own questions to StackOverflow. Be sure to hone in on the specific question you're trying to answer, and provide error messages and sample code as appropriate. I often find that, by the time I can articulate the question enough to post it, I've figured out my problem anyway.

   - There is a classical method of debugging called rubber duck debugging, which involves simply trying to explain your code/problem to an inanimate object (talking to pets works too). You'll usually be able to fix the problem if you just step back and think about how you would explain it to someone else!

4. **Documentation**: Python's documentation is pretty good: very thorough and usually contains helpful examples. The only problem is that it can occasionally be too detailed for beginners, and can sometimes be hard to navigate (where to find information is not immediately obvious in some situations). *Searching* the documentation (potentially through Google) is a good way to narrow down what you're looking for.

Resources

Python is an incredibly popular programming language and is especially accessible to new programmers. As such, there are **a lot** of resources available for learning how to program in Python. While this book will cover most of the details you need, below are a number of other effective resources below in case you need further help. *Note that you'll need to reference Python 3 materials.*

- Think Python (Downey): a free, friendly introductory textbook for learning Python. You should definitely read **Chapter 1**, which is a great

overview of programming in general.

- Python for Everybody (Severance): a remixed version of the above book, with a focus on Information Sciences. See in particular the interactive version. Chapter 1 of this book is also a very good read.
- Automate the Boring Stuff with Python (Sweigart): another good free textbook
- Python for Non-Programmers Index: an extensive list of resources and materials for learning to program with Python. The textbooks and interactive courses are all good options.
- Official Python Documentation
- Google's Python Style Guide

Specific resources for the material in this module:

- Python Basics (Sweigart)
- Variables, expressions, statements (Downey)
- Jupyter Notebook Documentation
- Jupyter Notebook Tutorial (Data Camp)

Chapter 5

Functions

This chapter will explore how to use and create **functions** in Python. Functions
are the primary form of *behavior abstraction* in computer programming, and are
used to structure and generalize code instructions. After considering a function
in an abstract sense, it will look at how to use built-in Python functions, how
to access additional functions by importing Python modules, and finally how to
write your own functions.

5.1   What are Functions?

In a broad sense, a **function** is a named sequence of instructions (lines of code)
that you may want to perform one or more times throughout a program. They
provide a way of *encapsulating* multiple instructions into a single "unit" that can
be used in a variety of different contexts. So rather than needing to repeatedly
write down all the individual instructions for "make a sandwich" every time
you're hungry, you can define a `make_sandwich()` function once and then just
call (execute) that function when you want to perform those steps.

In addition to grouping instructions, functions in programming languages like
Python also follow the mathematical definition of functions, which is a set of
operations (instructions!) that are performed on some **inputs** and lead to some
outputs. Functions inputs are called **arguments** or **parameters**, and we say
that these arguments are **passed** to a function (like a football). We say that a
function then **returns** an ouput for us to use.

5.2   Python Function Syntax

Python functions are referred to by name (technically they are values like any other variable). As in many programming languages, we **call** a function by writing the name of the function followed immediately (no space) by parentheses `()`. Inside the parentheses, we put the **arguments** (inputs) to the function separated by commas. Thus computer functions look just like mathematical functions, but with names longer than `f()`.

```python
# call the print() function, pass it "Hello world" value as an argument
print("Hello world")  # "Hello world"

# call the str() function, pass it 598 as an argument
str(598)  # "598"

# call the len() function, pass it "python" as an argument
len("python")  # 6 (the word is 6 letters long)

# call the round() function, pass it 3.1415 as the first arg and 2 as the second
# this is an example of a function that takes multiple (ordered) args
round(3.1415, 2)  # 3.14, (3.1415 rounded to 2 decimal places)

# call the min() function, pass it 1, 6/8, AND 4/3 as arguments
# this is another example of a function that takes multiple args
min(1, 6/8, 4/3)  # 0.75, (6/8 is the smallest value)
```

- *Note:* To keep functions and variables distinct, we try to always include empty parentheses `()` when referring to a function name. This does *not* mean that the function takes no arguments; it is just a useful shorthand for indicating that something is a function rather than a variable.

Some functions (such as `min()` or `print()`) can be passed as many arguments as you wish: `min()` will find the minimum of *all* the arguments, and `print()` will print *all* the arguments (in order), separated by spaces:

```python
# print() with 3 arguments instead of 1
print("I", "love", "programming")  # "I love programming"
```

Besides ordered **positional arguments**, functions may also take **keyword arguments**, which are arguments for specific function inputs. These are written like variable assignments (using the `=`, though without spaces around it), but within the function parameter list:

```python
# Use the `sep` keyword argument to specify the separator is '+++'
print("Hello", "World", sep='+++')  # "Hello+++World"
```

Keyword arguments are always optional (they have "default" values, like how the separator for `print()` defaults to a single space `' '`). The default values

are specified in the function documentation (e.g., for `print()`).

If you call any of these functions interactively (e.g., in an interactive shell or a Jupyter notebook), Python will display the **returned value**. However, the computer is not able to "read" what is written to the console or an output cell—that's for humans to view! If we want the computer to be able to *use* a returned value, we will need to give that value a name so that the computer can refer to it. That is, we need to store the returned value in a variable:

```python
# store min value in smallest_number variable
smallest_number = min(1, 6/8, 4/3, 5+9)

# we can then use the variable as normal, such as mathematical operations
twice_min = smallest_number * 2   # 1.5

# we can use functions directly in expressions (the returned value is anonymous)
number = .5 * round(9.8)   # 5.0

# we can even pass the result of a function as an argument to another!
# watch out for where the parentheses close!
print(min(2.0, round(1.4)))   # prints 1
```

5.2.1   Object Methods

In Python, all data values are **objects**, which are groups of data (called *attributes*) and behaviors—that is, information about the values and the *functions* that can be applied to that data. For example, a Person object may have a name (e.g., `"Ada"`) and some behavior it can do to that data (e.g., `say_name()`). Functions that are applied to an object's data are also known as **methods**. We say that a method is *called on* that object.

While we'll discuss objects in more much detail later, for now you just need to understand that some functions are called *on* particular values. This is done using **dot notation**: you write the name of the variable you wish to call the method on (i.e., apply the function to), followed by a period (dot) `.`, followed by the method name and arguments:

```python
message = "Hello World"

# call the lower() method on the message to make a lowercase version
# Note that the original string does not change (strings are immutable)
lower_message = message.lower()   # "hello world"

# call the replace() method on the message
western_message = message.replace("Hello", "Howdy")   # "Howdy World"
```

This is a common way of utilizing built-in Python functions.

- Note that dot notation is also used to access the **attributes** or **properties** of an object. So if a `Person` object has a `name` attribute, you would refer to that as `the_person.name`. In this sense, you can think of the *dot operator* as being like the possessive `'s` in English: `the_person.name` refers so "`the_person`*'s name*", and `the_person.say_name()` would refer to "`the_person`*'s* `say_name()` *action*".

5.3 Built-in Python Functions

As you may have noticed, Python comes with a large number of functions that are built into the language. In the above examples, we used the `print()` function to print a value to the console, the `min()` function to find the smallest number among the arguments, and the `round()` function to round to whole numbers. Similarly, each data type in the language comes with their own set of methods: such as the `lower()` and `replace()` methods on strings.

These functions are all described in the official documentation. For example, you can find a list of built-in functions (though most of them will not be useful for us), as well as a list of string methods. To learn more about any individual function as well as what functions are available, look it up in the documentation. You can also use the `help()` function to look up the details of other functions: `help(print)` will look up information on the `print()` function.

Important: "Knowing" how to program in a language is to some extent simply "knowing" what provided functions are available in that language. Thus you should look around and become familiar with these functions… but *do not* feel that you need to memorize them! It's enough to simply be aware "oh yeah, there was a function that rounded numbers", and then be able to look up the name and arguments for that function.

5.3.1 Modules and Libraries

While Python has lots of built-in functions, many of them are not immediately available for use. Instead, these functions are organized into **modules**, which are collections of related functions and variables. You must specifically load a module into your program in order to access it's functions; this helps to reduce the amount of memory that the Python interpreter needs to use in order to keep track of all of the functions available.

You load a module (make it available to your program) by using the `import` keyword, followed by the name of the module. This only needs to be done once per script execution, and so is normally done at the "top" of the script (in a Jupyter notebook, you can include an "importing" code cell, or import the module at the top of the cell in which you first need it):

```python
# load the math module, which contains mathematical functions
import math

# call the math module's sqrt() function to calculate square root
math.sqrt(25)  # 5.0, (square root of 25)

# print out the math modules `pi` variable
print(math.pi)  # 3.141592653589793
```

Notice that we again use **dot notation** to call functions of a particular module. Again, think of the `.` as a possessive `'s` ("*the* `math` *module's* `sqrt` *function*").

It is also possible to import only select functions or variables from a module by using the syntax `from MODULE import FUNCTION`. This will load the specific functions or variables into the *global scope*, allowing you to call them without specifying the module:

```python
# import the sqrt() function from the math module
from math import sqrt

# call the imported sqrt() function
sqrt(25)  # 5.0

# import multiple values by separating them with commas
from math import sin, cos, pi

sin(pi/2)  # 1.0
cos(pi/2)  # 6.123233995736766e-17; 0 but with precision errors

# import everything from math
# this is useful for module with long or confusing names
from math import *
```

The collection of built-in modules such as `math` make up what is called the *standard library* of Python functions. However, it's also possible to download and import additional modules written and published by the Python community—what are known as **libraries** or **packages**. Because many Python users encounter the same challenges, programmers are able to benefit from the work of other and *reuse* solutions. This is the amazing thing about the open-source community: people solve problems and then make those solutions available to others.

A large number of additional libraries are included with Anaconda. Anaconda also comes with a command-line utility `conda` that can be used to easily download and install new libraries. For example if you needed to install Jupyter, you could use the command `conda install jupyter` from the command line.

Python also comes with a command line utility called `pip` for ("pip installs

packages") that can similarly be used to easily download and install packages. Note that you may need to set up a virtual environment to effectively use `pip`; thus I recommend you rely on `conda` instead—though the Anaconda package has everything you will need for this course.

5.4 Writing Functions

Even more common than loading other peoples' functions is writing your own. Functions are the primary way that we *abstract* program instructions, acting sort of like "mini programs" inside your larger script. As Downey says:

> Their primary purpose is to help us organize programs into chunks that match how we think about the problem.

Any time you want to organize your thinking—or if you have a task that you want to repeat through the script—it's good practice to write a function to perform that task. This will make your program easier to understand and build, as well as limit repetition and reduce the likelihood of error.

The best way to understand the syntax for defining a function is to look at an example:

```python
# A function named `MakeFullName` that takes two arguments
# and returns the "full name" made from them
def make_full_name(first_name, last_name):
    # Function body: perform tasks in here
    full_name = first_name + " " + last_name

    # Return: what you want the function to output
    return full_name

# Call the make_ful_name function with the values "Alice" and "Kim"
my_name = make_full_name("Alice", "Kim")  # "Alice Kim"
```

Functions have a couple of pieces to them:

- **def**: Functions are defined by using the `def` keyword, which indicates that you are defining a function rather than a regular variable. This is followed by the **name** of the function, then a set of parentheses containing the arguments (Note that the `function_name(arguments)` syntax mirrors how functions are called).

  The line ends with a colon `:` which starts the function body (see below).

- **Arguments**: The values put between the parentheses in the function definition line are variables that *will contain* the values passed in as **arguments**. For example, when we call `make_full_name("Alice","Kim")`,

the value of the first argument (`"Alice"`) will be assigned to the first variable (`first_name`), and the value of the second argument (`"Kim"`) will be assigned to the second variable (`last_name`).

Importantly, we could have made the argument names anything we wanted (`name_first`, `given_name`, etc.), just as long as we then use *that variable name* to refer to the argument while inside the function. Moreover, these argument variable names *only apply* while inside the function (they are **scoped** to the function). You can think of them like function-specific "nicknames" for the values. The variables `first_name`, `last_name`, and `full_name` only exist within this particular function.

Note that a function may have no arguments, causing the parentheses to be empty:

```python
def say_hello():
  print("Hello world!")
```

You can give a function *keyword arguments* by assigning a default value to the argument in the function definition:

```python
# includes a single keyword argument
def greet(greeting = "Hello"):
  print(greeting + " world")

# call by assigning to the arg
greet(greeting="Hi")  # "Hi world"

# call without assigning an argument, using the default value
greet()  # "Hello world"
```

- **Body**: The body of the function is a **block** of code. The function definition ends with a colon `:` to indicate that it is followed by a *block*, which is a list of Python statements (lines of code). Which statements are part of the block are indicated by *indentation*: statements are indented by 4 spaces to make them part of that particular block. A block can contain as many lines of code as you want—you'll usually want more than 1 to make a function worthwhile, but if you have more than 20 you might want to break your code up into separate functions. You can use the argument variables in here, create new variables, call other functions... basically any code that you would write outside of a function can be written inside of one as well!

- **Return value**: You can specify what output a function produces by using the **return** keyword, followed by the value that you wish *your function* to return (output). A `return` statement will end the current function and *return* the flow of code execution to where ever this function was called from. Note that even though we returned a variable called `full_name`, that variable was *local* to the function and so doesn't exist outside of it;

thus we have to take the returned value and assign it to a new variable (as with `my_name = make_full_name("Alice", "Kim")`).

Because the `return` statement exits the function, it is almost always the last line of code in the function; any statements after a `return` statement will not be run!

- A function does not need to return a value (as in the `say_hello()` example above). In this case, omit the `return` statement.

We can call (execute) a function we defined the same way we called built-in functions. When we do so, Python will take the **arguments** we passed in (e.g., `"Alice"` and `"Kim"`) and assign them to the *argument variables*. Then the interpreter executes each line of code in the **function body** one at a time. When it gets to the `return` statement, it will end the function and return the given value, which can then be assigned to a different variable (outside of the function).

5.4.1 Doc Strings

We create new functions as ways of *abstracting* behavior, in order to make programs easier to read and understand. Thus it is important to be clear about the purpose and use of any functions you create. This can partially done through effective function and argument names: `def calc_rectangle_area(width, height)` is pretty self-explanatory, whereas `def my_func(a, b, c)` isn't).

Style requirement: Name functions as phrases starting with *verbs* (because function do things), and variables as nouns.

Nevertheless, in order to make sure that functions are as clear as possible, you should include **documentation** in the form of a *comment* that describes in plain English what a function does: its inputs (arguments) once, output (return value), and overall behavior. In Python we document functions by providing a **doc string**. This is a string literal (written in multi-line triple quotes `"""`) placed immediately below the function declaration:

```python
def to_celsius(degrees_fahrenheit):
    """Converts the given degrees (in Fahrenheit) to degrees in Celsius, and
        returns the result.
    """
    celsius = (degrees_fahrenheit - 32)*(5/9)
    return celsius
```

- Note that this string isn't assigned to a variable; it is treated as a valid statement because it is immediately below the function declaration.

- You can view the doc string by calling the `help()` function with your function's name as the parameter! In fact, when you call `help()` on any built-in function, what you are viewing is that function's doc string.

Doc strings should include the following information:

1. What the function does *from the perspective of someone who would call the function.* This should be a short (1-2 sentence) summary; don't describe the individual instructions that are executed, but an *abstraction* of the code's purpose.

   - If you mention variables, operations, functions, or control structures, your comment isn't at a high enough level of abstraction!

2. Descriptions of the expected arguments. Clarify any ambiguities in type (e.g., a number or a string) and units.

3. Description of the returned value (if any). Explain the meaning, type (e.g., number or string), etc.

It's good to think of doc strings as defining a "contract": you are specifying that "if you give the function this set of inputs, it will perform this behavior and give you this output.".

For simple methods you can build this information into a single sentence as in the above example. But for more complex functions, you may need to use a more complex format for your doc string. See the Google Style Guide for an example.

Resources

- Functions (Sweigart)
- Functions (Severance)
- Fruitful Functions (Downey)

Chapter 6

Logic and Conditionals

Programming involves writing instructions for a computer to execute. However, what allows computer programs to be most useful is when they are able to *decide which* instructions instructions to execute based on a particular situation. This is referred to as code branching, and is used to shape the flow of control of the computer code. In this chapter, you will learn how to utilize **conditional statements** in order to include control flow in your Python scripts.

6.1 Booleans

In addition to the basic data types `int`, `float`, and `str`, Python supports a *logical* data type called a **Boolean** (class `bool`). A boolean represents "yes-or-no" data, and can be exactly one of two values: `True` or `False` (note the capitalization). Importantly, these **are not** the Strings `"True"` and `"False"`; boolean values are a different type!

```
type(True)     # <class 'bool'>
type("True")   # <class 'str'>
type(true)     # NameError: name 'true' is not defined
               # e.g., no variable called `true`!
```

Fun fact: logical values are called "booleans" after mathematician and logician George Boole, who invented many of the rules and uses of this construction (called Boolean algebra).

Boolean values are most commonly the result of applying a **relational operator** (also called a **comparison operator**) to some other data type. Comparison operators are used to compare values and include: `<` (less than), `>` (greater than), `<=` (less-than-or-equal, written as read), `>=` (greater-than-or-equal, written as read), `==` (equal), and `!=` (not-equal).

```python
x = 3
y = 3.15

# compare numbers
x > y  # returns logical value False ("x is bigger than y" is a False statement)
y != x  # returns logical value True ("y is not-equal to x" is a True statement)

# compare x to pi (built-in variable)
y == math.pi  # returns logical value False

# compare strings (based on alphabetical ordering)
"cat" > "dog"  # returns False
```

Important == (two equals signs) is a comparison operator, but = (one equals sign) is the assignment operator!

Note that boolean variables should be named as *statements of truth*. Use words such as `is` in the variable name:

```python
is_early = True
is_sleeping = False
needs_coffee = True
```

6.1.1   Boolean Operators

In addition, boolean values support their own operators (called **logical operators** or **boolean operators**). These operators are applied to boolean values and produce boolean values, and allow you to make more complex *boolean expressions*:

- **and** (conjunction) produces `True` if both of the operands are `True`, and `False` otherwise
- **or** (disjunction) produces `True` if *either* of the operands are `True`, and `False` otherwise
- **not** (negation) is a unary operator that produces `True` if the operand is `False`, and `False` otherwise

```python
x = 3.1
y = 3.2

# Assign bool values to variables
x_less_than_pi = x < math.pi  # True
y_less_than_pi = y < math.pi  # False

# Boolean operators
x_less_than_pi and y_less_than_pi  # False
```

```python
x_less_than_pi or y_less_than_pi  # True

# This works because Python is amazing
x < math.pi < y  # True

# Assign str value to a variable
pet = "dog"

# It is NOT the case that pet is "cat"
not pet == "cat"  # True

# pet is "cat" OR "dog"
pet == "cat" or pet == "dog"  # True

# This doesn't work (operators are applied left-to-right; check the type
# at each step!)
# See "short-circuiting" below, as well as http://stackoverflow.com/a/19213583
pet == "cat" or "dog"  # "dog"
```

Because boolean expressions produce *more* booleans, it is possible to combine these into complex logical expressions:

```python
# given two booleans P and Q
P = True
Q = False

P and not Q  # True
not P and Q  # False
not (P and Q)  # True
(not P) or (not Q)  # True
```

The last two expressions in the above example are equivalent logical statements for *any* combination of values for `P` and `Q`, what is known as De Morgan's Laws. Indeed, many logical statements can be written in multiple equivalent ways.

Finally, note that when using an `and` operator, Python will **short-circuit** the second operand and never evaluate it if the first is found to be `False` (after all, if the first operand isn't `True` there is no way for the entire `and` expression to be!)

```python
x = 2
y = 0
x == 2 and x/y > 1  # ZeroDivisionError: division by zero
x == 3 and x/y > 1  # no error (short-circuited)

# Use a "guardian expression" (make sure y is not 0)
# to avoid any errors
```

```
y != 0 and x/y > 1  # no error (short-circuited)
```

The reason this works is because when the Python interpreter reads the expression `P and Q`, it produces `Q` if `P` is `True`, and `P` otherwise. After all if `P` is `True`, then the overall truth of the expression is dependent entirely on `Q` (and thus that can just be returned). Similarly, if `P` is `False`, then the whole statement is False (which is the value of `P`, so just return that!)

6.2 Conditional Statements

One of the primary uses of Boolean values (and the *boolean expressions* that produce them) is to control the flow of execution in a program (e.g., what lines of code get run in what order). While we can use *functions* are able to organization instructions, we can also have our program decide which set of instructions to execute based on a set of conditions. These decisions are specified using **conditional statements**.

In an abstract sense, an conditional statement is saying:

```
IF something is true
    do some lines of code
OTHERWISE
    do some other lines of code
```

In Python, we write these conditional statements using the keywords **if** and **else** and the following syntax:

```
if condition:
    # lines of code to run if condition is True
else:
    # lines of code to run if condition is False
```

The **condition** can be any boolean value (or any expression that evaluates to a boolean value). Both the **if** statement and **else** clause are followed by a colon **:** and a **block**, which is a set of *indented* statements to run (similar to the blocks used in functions). It is also possible to *omit* the `else` statement and its block if there are no instructions to run in the "otherwise" situation:

```
porridge_temp = 115  # temperature in degrees F

if porridge_temp > 120:
    print("This porridge is too hot!")
else:
    print("This porridge is NOT too hot!")

too_cold = porridge_temp < 70  # a boolean variable
if too_cold:
```

```
    print("This porridge is too cold!")

# This line is outside the block, so is not part of the conditional
# (it will always be executed)
print("Time for a nap!")
```

Blocks can themselves contain *nested* conditional statements, allowing for more complex decision making. Nested `if` statements are indented twice (8 spaces or 2 tabs). There is no limit to how many "levels" you can nest; just increase the indentation each time.

```
# nesting example
if outside_temperature < 60:
    print("Wear a jacket")
else:
    if outside_temperature > 90:
        print("Wear sunscreen")
    else:
        if outside_temperature == 72:
            print("Perfect weather!")
        else:
            print("Wear a t-shirt")
```

Note that this form is nesting is also how you can use conditionals inside of functions:

```
def flip(coin_is_heads):
    if coin_is_heads:
        print("Heads you win!")
    else:
        print("Tails you lose.")
```

If you consider the above nesting example's logic carefully, you'll notice that many of the "branches" are **mutually exclusive**: that is, the code will choose only 1 of 4 different clothing suggestions to print. This can be written in a cleaner format by using an **elif** ("else if") clause:

```
if outside_temperature < 60:
    print("Wear a jacket")
elif outside_temperature > 90:
    print("Wear sunscreen")
elif outside_temperature == 72:
    print("Perfect weather!")
else:
    print("Wear a t-shirt")
```

In this situation, the Python interpreter will perform the following logic:

1. It first checks the `if` statement's condition. If that is `True`, then that branch is executed and the rest of the clauses are skipped.
2. It then checks each `elif` clause's condition *in order*. If one of them is `True`, then that branch is executed and the rest of the clauses are skipped.
3. If *none* of the `elif` clauses are `True`, then (and only then!) the `else` block is executed.

This *ordering* is important, particularly if the conditions are not in fact mutually exclusive:

```python
if porridge_temp < 120:
    print("This porridge is not too hot!")
elif porridge_temp < 70:
    # unreachable statement! the condition will never be both checked and True
    print("This porridge is too cold!")

# contrast with:
if porridge_temp < 120:
    print("This porridge is not too hot!")
if porridge_temp < 70:  # a second if statement, unrelated to the first
    print("This porridge is too cold!")
# both print statements will execute for `porridge_temp = 50`
```

See also the resources at the end of the chapter for explanations with logical diagrams and flowcharts.

6.2.1   Designing Conditions

Relational operators all have logical opposites (e.g., `==` is the opposite of `!=`; `<=` is the opposite of `>`), and boolean expressions can include negation. This means that there are many different ways to write conditional statements that are *logically equivalent*:

```python
# these two statements are equivalent
if x > 3:
    print("big X!")

if 3 < x:
    print("big X!")
```

The second example is known as a Yoda condition; in Python you should use the former version (variable parts of the condition on the left).

Thus you should follow the below guidelines when writing conditionals. These will let you produce more "idiomatic" code which is cleaner and easier to read, as well as less likely to cause errors.

- Avoid checks for mutually exclusive conditions with an `if` and `elif`. Use the `else` clause instead!

```python
# Do not do this!
if temperature < 50:
    print("It is cold")
elif temperature >= 50:  # unnecessary condition
    print("It is not cold")

# Do this instead!
if temperature < 50:
    print("It is cold")
else:
    print("It is not cold")
```

- Avoid creating redundant boolean expressions by comparing boolean values to `True`. Instead, use an effectively named variable.

```python
# Do not do this!
if is_raining == True:  # unnecessary comparison
    print("It is raining!")

# Do this instead!
if is_raining:  # condition is already a boolean!
    print("It is raining!")
```

Note that this gets trickier when trying to check for `False` values. Consider the following equivalent conditions:

```python
# I believe this is the cleanest option, as it reads closet to English
if not is_raining:
    print("It is not raining!")

# This is an acceptable equivalent, but prefer the first option
if is_raining == False:
    print("It is not raining!")

# Use one of the above options instead
if is_raining != True:
    print("It is not raining!")

# This can be confusing unless your logic is explicitly based around the
# ABSENCE of some condition
if is_not_raining:
    print("It is not raining!")
```

Overall, try to develop the simplest, most straightforward conditions you can. This will make sure that you are able to think clearly about the program's

control flow, and help to clarify your own thinking about the program's logic.

6.3   Determining Module or Script

As discussed in Chapter 5, it is possible to define Python variables and functions in `.py` files, which can be run as stand-alone scripts. However, these files can also be imported as modules, allowing you to access their variables and functions from *another* script file—similar to how you imported the `math` module. Thus all Python scripts have two "modes": they can be used as executable scripts (run with the `python` command at the command line), or they can be imported as modules (libraries of variables and functions, using the `import` keyword in the script).

When the `.py` script is run as an executable top-level script, you often want to perform special instructions (e.g., call specific functions, prompt for user input, etc). You can use a special *environmental* variable called `__name__` to determine whether a script is the "main" script that is being run, and `if` so execute those instructions:

```python
if __name__ == "__main__":
    # execute only if run as a script
    print("This is run as a script, not a module!")
```

The `__` in the variable names and values are a special naming convention used to indicate *to the human* that these values are **internal** to the Python language and so have special meaning. They are otherwise normal variables names.

It is best practice to define all the functions your script will utilize, and then use the "is run as a script" check to determine whether or not to execute those functions—this will allow you to easily reuse you code in a different task without adding any side effects!

Resources

- Conditional Execution (Downey)
- Conditionals (Severance)
- Flow Control (Sweigart) (first half)

Chapter 7

Iteration and Loops

One of the main benefits of using computers to perform tasks is that computers never get tired or bored, and so can do the same thing over and over and over and over and over and over again. This is a process known as **iteration**. Iteration represents another form of *control flow* (similar to conditionals), and is specified in a program using a set of statements called **loops**. In this chapter, you will learn the basics of writing loops to perform iteration; more advanced iteration concepts will be covered in later chapters.

7.1 While Loops

Loops are **control structures** (similar to `if` statements) that allow you to perform iteration. These statements specify that a *block* of code should be executed repeatedly—the block is executed statement by statement, and then the flow of control "loops" back to the top to execute the statements again. Programming languages such as Python support a number of different kinds of loops, which differ primarily in how they determine whether or not to repeat the block (though this difference is reflected in the syntax).

In programming, the most basic *control structure* used for iteration is known as a **while loop**, which is used for "indefinite iteration". A Python while loop has the following structure:

```python
while condition:
    # lines of code to run if the condition is True
```

This construction looks much like an `if` statement, and is similar in many regards. As with an `if` statement, the **condition** can be any boolean value (or any expression that evaluates to a boolean value), and it determines whether or not the loop's *block* will be executed.

In order to understand how a while loop influences the flow of program control, consider a more concrete example:

```python
count = 5
while count > 0:
    print(count)
    count = count - 1


print("Blastoff!")
```

In this example, when the interpreter reaches the `while` statement, it first checks the condition (`count > 0`, where `count` is 5). Finding that condition to be `True`, the interpreter then executes the block, printing the `count` and then *decrementing* that variable. Once the block is executed, the interpreter **loops back to the `while` statement** and rechecks the condition. Finding that `count` (4) is still greater than 0, it executes the block *again* (causing `count` to decrement again). This continues until the interpreter loops to the `while` statement and finds that `count` has reached 0, and thus is no longer greater than 0. Since the condition is now `False`, the interpreter does *not* enter the loop, and proceeds to the following statement ("Blastoff!").

Importantly, the condition is only checked when the block is *about* to execute: both at the "start" of the loop, and then at the beginning of each subsequent iteration. Having the condition become `True` in the middle of the block (e.g., temporarily) will have no impact on the control flow. It is also possible for the interpreter to "never enter" the loop if the condition is not initially `True`.

7.1.1   Counting and Loops

The above example also demonstrates how to use a "counter" to determine whether or not the loop has run a sufficient number of times: this is known as a **loop control variable (LCV)**. The "standard" counting loop looks like:

```python
count = 0               # 1. initialize the counter
while count < 100:      # 2. check if the counter has reached its target
    print(count)        # 3. do some work (this may be multiple statements)
    count = count + 1   # 4. update the counter
```

In order for a counted loop to work properly, you need to be careful about steps 2 and 4: the condition and the update.

First, recall that the condition is *whether to run the loop*, not *whether to stop*:

```python
count = 0
while count == 100:   # bad condition!
    print(count)
    count = count + 1
```

In this case, the condition is not initially `True`, so the interpreter never enters the loop.

When writing conditions, think *"do we keep going"* rather than *"are we there yet?"*. In loops (as in life), the journey is more important than the destination!

Second, consider what happens if you forget to update the loop control variable:

```python
count = 0
while count < 100
    print(count)
    # no counter update
```

In this case, the interpreter checks that `count` (0) is less than 100, then runs the loop. Then checks that `count` (still 0) is less than 100, then runs the loop. Then checks that `count` (*still* 0) is less than 100, then runs the loop...

This is known as an **infinite loop**: the loop will run forever, never being able to "break out" and reach the next statement.

If you hit an infinite loop in Jupyter Notebook, use `Kernel > Interrupt` to break it and try again. If running a `.py` file on the command line, use `Ctrl-C` to cancel the script.

There are lots of ways to accidentally produce an infinite loop, including:

1. Having a condition that is "too exact" can cause the loop control variable to "miss" a particular breaking value:

   ```python
   count = 0
   while count != 100:  # if we aren't yet at 100
       print(count)
       count = count + 3  # this will never equal 100
   ```

   Thus it is always safer to use **inequalities** (e.g., `<` or `>`) when writing loop conditions.

2. Resetting the counter in the body of the loop can cause it to never reach its goal:

   ```python
   count = 0
   while count < 100:
       count = 0  # this resets the count!
       print(count)
       count = count + 1
   ```

The best way to catch these errors is to "play computer": pretend that you are the compiler, and go through each statement one by one, keeping track of the loop control variable (writing its value down on a sheet of paper does wonders). This will help you be able to "trace" what your program is doing and catch any bugs there may be.

7.1.2   Nested Conditionals and Sentinels

As with `if` statements, the block for a `while` loop can contain any valid Python statements, including `if` statements or even other loops (called a "nested loop"). Control statements such as `if` are indented an extra step (4 spaces or 1 tab):

```python
# flip a coin until it shows up heads
still_flipping = True
while still_flipping:
    flip = random.randint(0,1)
    if flip == 0:
        flip = "Heads"
    else:
        flip = "Tails"
    print(flip)
    if flip == "Heads":
        still_flipping = False
```

In this example, the `still_flipping` boolean variable acts as the *loop control variable*, as it determines whether or not the loop is repeated. Using a boolean as a LCV is known as using a **sentinel variable**. A sentinel (guard) variable is used to control whether or not the program flow gets out of the loop: as long as the sentinel is `True`, the loop continues to run. Thus the loop can be "exited" by assigning the sentinel to be `False`. This is particularly useful when there may be a complex set of conditions that need to be met before the program can carry on.

(It is of course possible to design a sentinel such as `done_flipping`, and then have the `while` condition check that the sentinel is **not** `True`. This may be useful depending on how you've structured the algorithm. In either case, be sure that your sentinels are named carefully and accurately reflect the information conveyed by the variable!)

7.2   For Loops

If you look back at the basic counting loop, you'll notice that tracking the `count` loop control variable can be problematic. It is easy to forget the update statement (`count = count + 1`), and the the `count` variable itself acts as an extra "global" (or "less local") variable that the interpreter needs to keep track of.

To avoid these problems, we can instead use a different kind of loop called a **for loop**. A for loop is used for "definite iteration": when we want execute a loop a specific number of times. The basic Python for loop has the following structure:

```python
for local_variable in range(maximum):
    # lines of code to run for each number
```

For example, the basic counting while loop example could be rewritten using a simpler for loop:

```python
for count in range(100):
  print(count)
```

The `range()` function returns a value representing a sequence of numbers. It is an example of a **sequence** data structure, which is a way of representing multiple data items in a single variable. Sequences will be discussed more in later chapters (including the most common type of sequence, a **list**). The `range()` function can be called with different arguments depending on the range and spread of numbers you wish to use:

```python
# numbers 0 to 10 (not inclusive)
# produces [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
range(10)

# numbers 1 (inclusive) through 11 (not inclusive)
# produces [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
range(1,11)

# numbers 0 (inclusive) through 10 (not inclusive), skipping by 3
# produces [0, 3, 6, 9]
range(0, 10, 3)
```

Thus `for count in range(100)` can be read as "for *each* number (called count) from 0 to 100".

For loops may more properly be thought of as **"for each"** loops; they are used to go through the items in a collection (e.g., each number in a `range`), executing the loop body once *for each* item. The **local_variable** (e.g., count) in a for loop is *implicitly* assigned the value of the "current" item in the collection (e.g., which number in the range you're on) at each iteration.

- For ranges, this basically means that the for loops keeps track of the current iteration; but the idea of this local variable will be more important as we introduce additional collection types.

7.2.1 Difference from While Loops

The main difference between while loops and for loops is:

> while loops are used for **indefinite** iteration; for loops are used for **definite** iteration.

While loops are appropriate when the interpreter doesn't know *in advance* (before the loop starts) how many times the loop block will be executed: the loop does not have a definite number of iterations. On the other hand, a for loop is appropriate when the interpreter does know *in advance* (before the loop starts) how many times the block will be executed—a definite number of iterations. Note that the number of iterations may be a variable so not determined until runtime; however, the value of that variable will still be known when the `for` statement is executed.

All iteration can be written with a while loop; but when performing definite iteration, it is easier, faster, and more idiomatic to use a for loop!

7.3   Iterating over Files

For loops can be used to iterate through any collection (technically any "iterable" type). One of the more useful collections when working with data is external **files** (e.g., text files). Files can be treated as a collection or sequence of *lines* (each divided by a `\n` newline character), and thus Python can "read" a text file using a for loop to iterate over the lines of text in the file.

In order to read or write text file data, you use the built-in `open()` function, passing it the *path* to the file you wish to access. This function will then return an object representing that particular file (e.g., it's location on the disk), with methods that you can use to read from and write to it.

```python
my_file = open('myfile.txt')  # open the file

for line in my_file:
    print(line)  # print each line in the file
```

Remember to **always** use *relative* paths. Note that when using a Jupyter Notebook, the "current working directory" is the directory in which you ran the `jupyter notebook` command to start the server.

Once you have opened a file, you can use a for loop to iterate through its line (as in the example above). You can also use a while loop, calling the `readline()` method *on* the file in order to read a single line at a time. Note that once you've looped through a file, that file has been "read" and will not be able to be looped through again unless you "re-open" it. In general, you should only read through a file once in any program you write!

After a file has been opened, it will remain "open" until it is explicitly closed. The recommended way to do this is by using the `with` keyword, which will make sure that the file is closed either when it is done being read or if some kind of error occurs:

```python
with open('myfile.txt') as my_file:
    for line in my_file:
        print(line)
```

It is also possible to write out content to a file. To do this, you need to open the file with "write" access (allowing the program to write to and modify it) by passing w as the second argument to the `open()` function. You can then use the `write()` method to "print" text to the file:

```python
# "open" the file with "write" access
my_file = open('myfile.txt', 'w')

my_file.write("Hello world!\n")
my_file.write("It's a mighty fine morning\n")
```

Note that unlike the `print()` function, `write()` does not include a line break at the end of each method call; you need to add those yourself!

7.3.1 Try/Except

File operations rely on a context that is external to the program itself: namely, that the file you wish to open actually exists at the location you specify! But that may not be the case—particularly if which file to open is specified by the user:

```python
filename = input("File to open: ")  # ask user which file to open

my_file = open(filename)  # "open" the file

for line in my_file:
    print(line)
```

If the user provides a bad file name, your program will encounter an error *through no fault of your own as a programmer*:

```
$ python script.py
File to open: neener neener
Traceback (most recent call last):
  File "script.py", line 4, in <module>
    my_file = open(filename)
FileNotFoundError: [Errno 2] No such file or directory: 'neener neener'
```

Since it's possible for the user to make a mistake, we could like the program not to simply fail with an error, but to instead be able to "recover" and keep running (e.g., by asking the user for a different file name). We can perform this kind of **error handling** by utilizing a **try** statement with an **except** clause:

```python
filename = input("File to open: ")

try: # this might break (not our fault)
    my_file = open(filename)
    for line in my_file:
        print(line)
except:  # catching FileNotFoundError
    print("No such file")
```

A `try` statement acts somewhat similar to an `if` statement; however, the `try` statement checks to see if any errors occur *within its block*. If such an error occurs, rather than the program crashing, the interpreter will *immediately* jump to the `except` class and execute that block, before continuing on with the rest of the program.

- Note that this is an exception to the general rule that conditions are only checked at the start of a block; a `try` block effectively tells the computer to keep an eye out for any errors ("try this, but it might break"), with the `except` clause specifying what to do if such an error occurs.

- The `with` keyword described above include a `try/except` that will make sure the file is closed if there is an error.

`try` statements are used when a program may hit an error that is *not caused by programmer's code, but by an external input* (e.g., from a user or a file). You should not use a `try` statement to fix broken program logic or invalid syntax: instead, you should fix those problems directly!

Resources

- Iterations (Downey)
- Flow Control (Sweigart) (second half)
- Iteration (Severance)
- Files (Downey)
- Files and File Paths (Sweigart)

Chapter 8

Lists and Sequences

This chapter will cover using **lists** in Python, which is a data type representing a *sequence* of data values (similar to how a `string` is a sequence of letters, and a `range` is a sequence of numbers). A list is a fundamental data type in Python, and key to writing almost all practical programs. Lists are used to store and organize large sets of data (and computer programs usually deal with *lots* of data). This chapter covers how to create, access, and utilize lists to automate the processing of larger amounts of data.

8.1 What is a List?

A list is a **mutable**, **ordered** sequence of values that are all stored in a single variable. For example, you can make a vector `names` that contains the strings "Sarah", "Amit", and "Zhang", or a vector `one_to_hundred` that stores the numbers from 1 to 100. Each value in a list is referred to as an **element** of that list; thus your `names` list would have 3 elements: `"Sarah"`, `"Amit"`, and `"Zhang"`.

Lists are written as literals inside *square brackets* (`[]`), with each element in the list separated by a *comma* (`,`):

```python
# A list of names
names = ["Sarah", "Amit", "Zhang"]

# A list of numbers (lists can contain "duplicate" values)
numbers = [1, 2, 2, 3, 5, 8]

# Lists can contain different types (including other lists!)
things = ["raindrops", 2, True, [5, 1, 1]]
```

```python
# Lists can be empty (with no elements)
empty = []
```

Style Requirement: List variables should be named using **plurals** (names, numbers, etc.), because lists hold multiple values!

Other sequences (such as strings or ranges) can be converted into lists by using the built-in `list()` function:

```python
list("hello")  # ['h', 'e', 'l', 'l', 'o']
list(range(1,5))  # [1, 2, 3, 4]
```

8.2 List Indices

You can refer to individual elements in a list by their **index**, which is the number of their position in the list. Lists are *zero-indexed*, which means that positions are counted starting at 0. For example, in the list:

```python
vowels = ['a', 'e', 'i', 'o', 'u']
```

The `'a'` (the first element) is at *index* 0, `'e'` (the second element) is at index 2, and so on. This also means that the last element can be found at the index `length_of_list - 1`. This pattern is why values like `range(5)` *include* 0 but *exclude* the last 5.

You can retrieve an element from a list using **bracket notation**: you refer to the element at a particular index of a list by writing the name of the list, followed by square brackets (`[]`) that contain the index of interest:

```python
names = ["Sarah", "Amit", "Zhang"]

# Access the element at index 0
name_first = names[0]
print(name_first)  # Sarah

# Access the element at index 2
name_third = names[2]
print(name_third)  # Zhang

# Accessing an index not in the list will give an error
name_fourth = names[3]  # IndexError!

# Negative indices count backwards from the end
name_last = names[-1]  # Zhang
name_second_to_last = names[-2]  #Amit
```

The value inside the square brackets can any expression that resolves to an integer, including variables:

```python
names[1+1]  # "Amit"

last_index = len(names) - 1  # last index is length of list - 1
names[last_index]  # "Zhang"

# Don't forget to subtract one from the length!
names[len(names)]  # IndexError!
# Using an index of `-1` is a better way to get the last element
```

It is possible to select multiple, *consecutive* elements from a list by specifying a **slice**. A slice is written as the starting and ending indices separated by a colon (**:**); the starting index is included and the ending index is *excluded*. For example:

```python
letters = ['a', 'b', 'c', 'd', 'e', 'f']

# Indices 1 through 3 (non-inclusive)
letters[1:3]  # ['b', 'c']

# Indices 3 to the end (inclusive)
letters[3:]  # ['d', 'e', 'f']

# Indices up to 3 (non-inclusive)
letters[:3]  # ['a', 'b', 'c']

# Indices 2 to (2 from end) (non-inclusive)
letters[2:-2]  # ['c', 'd'], the `e` is excluded

# All the indices. This produces a new list with the same contents!
letters[:]
```

The most important thing to note about lists is that an indexed reference to a list element (e.g., names[0]) is effectively a **variable in its own right**: you can think of names[0], names[1], and names[2] as being equivalent to having variables names_0, names_1, names_2 (each of which has its own value). Lists effectively provide a "shortcut" for having lots and lots of variables that are all related; instead you can "collect" those variables into a list!

Because list references are variables, they can be used anywhere that a "normal" variable can be. In particular, this means that variables can be assigned to them, allowing the list to be **mutated** (changed):

```python
# A list of school supplies
school_supplies = ["Backpack", "Laptop", "Pen"]
```

```python
# Replace "Pen" with "Pencil"
school_supplies[2] = "Pencil"

print(school_supplies)  # ['Backpack', 'Laptop', 'Pencil']

# You can only assign values to "variables" that exist in the list!
school_supplies[3] = "Paper"  # IndexError!
```

Just as with variables: if the list index (e.g., `my_list[index]`) is on the *left* side of an assignment, it means the **variable** (which "slot" in the list). If it is on the *right* side of an assignment, it means the **value** (which element is in that slot).

```python
letters = ['a', 'b', 'c']

letters[0] = 'q'  # `letters[0]` is the variable we assign to
first_letter = letters[0]  # `letters[0]` is the value assigned
```

Finally, note that **strings** are also *indexed sequences* of characters. Thus you can use *bracket notation* to refer to individual letters, and many string methods utilize the index in their arguments:

```python
message_str = "Hello world"
message_str[1:5]  # "ello"

# find the index of the 'w'
message_str.find('w')  # 6
```

8.3   List Operations and Methods

Lists support a number of different operations and methods (functions), some of which are demonstrated below:

```python
# Addition (+) to combine lists
['a','b'] + [1,2]  # ['a', 'b', 1, 2]

# Multiplication (*) performs multiple additions
[1,2] * 3  # [1, 2, 1, 2, 1, 2]

# A sample list
s = ['a', 'b', 'c', 'd']

# Add value to end of list
s.append('x')  # add 'x' at end
```

```
# Add value in middle of list
s.insert(2, 'y')  # put 'y' at index 2 (everyone else shifts over)

# Remove from end of list ("pop off")
s.pop()  # removes and returns the last item (`x`)

# Remove from middle
s.pop(2)  # removes and returns item at index 2 (`y`)

# Remove specific value
s.remove('c')  # remove the first 'c' in the list; nothing returned

# Remove all elements
s.clear()
```

Note that list methods such as `append()` or `clear()` usually **mutate** the existing list value and then return `None`. In comparison, string methods (such as `lower()` or `replace()`) will return a *different* string value. This is because lists can be changed, but strings cannot (they are *immutable*).

When comparing lists using a *relational operator* (e.g., `==` or `>`), the operation is applied to the lists **member-wise**: each element in the first list operand is compared to the element *at the same index* in the second list operand. If the comparison is `True` for *each* pair of elements, then the entire expression is `True`. In practice, this means that (a) you can use `==` to compare the contents of lists, and (b) a list is "smaller" than another if its first element is smaller than the other's first element.

But be careful: just because two lists have the same contents (are `==`) does not mean that they are *the same list*! In particular, two lists can be *different objects* (values) but still have the same contents. In Python, you can test whether two values are actually *the same value* (as opposed to having the same content) using the **is** operator.

```
# With strings, literals are shared (because they cannot be mutated)
str_a = "banana"  # `a` labels string literal "banana"
str_b = "banana"  # `b` labels string literal "banana"

# Both variables label the same (literal) value
str_a is str_b  # True

# With lists, each list created is a different object!
list_a = [1,2,3]  # `a` labels a new list [1,2,3]
list_b = [1,2,3]  # `b` labels a new list [1,2,3]

a == b  # True, have same values as contents
a is b  # False, are two different objects
```

```python
list_c = list_a  # `c` labels the value that `a` labels
a is c  # True, both are the same object

# Modify the list!
list_a[0] = 10
print(list_b[0])  # 1 (is a different list, so not changed)
print(list_c[0])  # 10 (is the same list, so is changed)
```

Keeping track of whether a *new* list has been created is particularly important when using lists as **arguments to functions**. Function arguments are local variables that are *assigned* the passed value—if this value is a list, then the assigned variable will refer to the same list, and any modifications to the argument will affect the value outside of the function:

```python
# A version of a function that modifies the list
def delete_first(a_list):
    a_list.pop(0)  # modifies the given value

letters = ['a','b','c']
delete_first(letters)  # call function
print(letters)  # ['b', 'c'], variable is changed

# A version of a function that does not modify the list
def delete_first(a_list):
    a_list = a_list[1:]  # create new local variable (replacing old local var)

letters = ['a','b','c']
delete_first(letters)  # call function
print(letters)  # ['a', 'b', 'c'], variable is not changed
```

8.4   Lists and Loops

As with other iterable sequences like strings and files, it is possible to iterate through the contents of a list by using a **for loop**:

```python
numbers = [3.98, 8, 10.8, 3.27, 5.21]

for element in numbers:  # `number` would be a better local variable name
    print(element)

# This will not let you modify the list, since the "number" variable is local
for number in numbers:
  number = round(number, 1)
```

```
print(numbers)  # [3.98, 8, 10.8, 3.27, 5.21], not rounded
```

In order to *modify* the list while iterating through it, you need a way to refer to the element you want to change. Since you refer to elements by their *index*, a better solution is to instead iterate through the *range of indices* rather than through the elements themselves:

```
numbers = [3.98, 8, 10.8, 3.27, 5.21]

for i in range(len(numbers)):  # `i` for "index"
    print(numbers[i])  # refer to elements by index

# Can now modify the list
for i in range(len(numbers)):
  numbers[i] = round(numbers[i], 1)  # change value at index to rounded version

print(numbers)  # [4.0, 8, 10.8, 3.3, 5.2], rounded!
```

Note that this process of applying some change to each element in a list is known as a **mapping** (each value "maps" or goes to some transformed version of itself). We will discuss mapping more in a later module.

8.5 Nested Lists

As noted at the start of the chapter, lists elements can be of **any** data type (and any *combination* of data types)—including other lists! These "lists of lists" are known as **nested lists** or **2-dimensional lists** (or *3d-lists* for a "list of lists of lists", etc). Nested lists are most commonly used to represent information such as *tables* or *matrices*.

Nested lists work exactly like normal lists—the elements just happen to themselves be indexable (like strings!):

```
# A list of different dinners available at a fancy party
# This list has 4 elements, each of which is a list of 3 elements
# (the indentation is just for human readability)
dinner_options = [
    ["chicken", "mashed potatoes", "mixed veggies"],
    ["steak", "seasoned potatoes", "asparagus"],
    ["fish", "seasoned rice", "green beans"],
    ["portobello steak", "seasoned rice", "green beans"]
]

len(dinner_options)   # 4
fish_option = dinner_options[2]  # ["fish", "seasoned rice", "green beans"]
```

```python
# Because `fish_option` is a list, we can reference its elements by index
print(fish_option[0])  # "fish"
```

In this example, `fish_option` is a variable that refers to a list, and thus its elements can be accessed by index using bracket notation. But as with any operator or function, it is also possible to use bracket notation on an *anonymous value* (e.g., a literal value that has not been assigned to a variable). That is, because `dinner_options[2]` is a list, we can use bracket notation refer to an element of that list without assigning it to a variable first:

```python
# Access the 2th element's 0th element
dinner_options[2][0]  # "fish"
```

This "pair of brackets" notation allows you to easily access elements within nested lists. This is particularly useful for 2d-lists that represent *tables* as a list of "rows" (often data records), each of which is a list of "column cells" (often data features):

```python
# A simple table of values
table = [ ['aa','ab','ac','ad'],
          ['ba','bb','bc','bd'],
          ['ca','cb','cc','cd'] ]

row = 1  # cells starting with 'b'
col = 3  # cells ending with 'd'
table[row][col]  # "bd", the cell at (row, col)
```

Note that you often use *nested for loops* to iterate through a *nested list*:

```python
for i in range(len(table)):  # go through each row (with index)
    for j in range(len(table[i]))  # go through each col of that row (with index)
        print(table[i][j])  # access element at ith row, jth column
```

We use a `j` for the index of the nested loop, because the `i` for "index" was already taken! `row` and `col` are also excellent local variable names.

8.6 Tuples

While lists are *mutable* (changeable) sequences of data, **tuples** represent ***immutable*** sequences of data. These are useful if you want to enforce that a data value won't be changed, such as for a function argument (or a dictionary key; see Chapter 10). Indeed, many built-in Python functions utilize tuples.

Tuples are written as *comma-separated sequences* of values. They are often placed inside parentheses for clarity (to help indicate the start and end of the tuple values):

```python
letters_tuple = ('a', 'b', 'c')
print(letters_tuple)  # ('a', 'b', 'c')

# Also a tuple (without parentheses)
numbers_tuple = 1, 2, 3
print(numbers_tuple)  # (1, 2, 3)

# A tuple representing a person's name, age, and whether they are hungry
# Tuple values have _implied_ meanings, which should be explained in comments
hungry_person = ('Ada', 28, True)

# In English, tuples may be named based on the number of elements
triple = (1,2,3)
double = (4,5)
single = (6,)  # extra comma indicates is a tuple, not just the int `6`
empty = ()  # an empty expression is a tuple!
type(())  # <class 'tuple'> (type of empty expression)
```

It's important to note that while you often write tuples in parentheses (and they are printed in parentheses), it is the **commas** that makes a sequence of literals into a tuple. The parentheses act just like like they do in mathematical expressions—they are only necessary to clarify ambiguity in the order-of-operations. You will find that some "idiomatic" expressions using tuples forgo the parentheses, making the syntax look more magical than it is!

Elements in tuples can be accessed by **index** using **bracket notation**, just like the elements in lists. However, tuples *cannot be modified*, so you cannot assign a new value to an index in a tuple:

```python
letter_triple = ('a','b','c')
print(letter_triple[0])  # 'a'
print(letter_triple[1:3])  # ('b','c'), a tuple

letter_triple[0] = 'q'  # TypeError!
```

Tuples can be compared using *relational operators* just like lists, and have the same "member-wise" comparison behavior described above. This makes it easy to order the immutable tuples just like you would order numbers or strings.

Finally, tuples provide one additional useful feature. The Python interpreter uses tuples to perform **multiple assignments**, where you assign multiple values to multiple variables in a single statement. You have already been able to assign multiple, comma-separated values to a single *tuple* variable (a process called **packing**). But Python also supports having a single sequential value (e.g., a *tuple*) be assigned to multiple, comma-separate variables (a process called **unpacking**)!

```python
triple = 1, 2, 3  # assign multiple values to single variable (packing)
print(triple)  # (1, 2, 3)

x, y, z = (1,2,3)  # assign single tuple value to multiple variables (unpacking)
print(x)  # 1
print(y)  # 2
print(z)  # 3

# The VALUE in this statement is evaluated as a tuple, and then is assigned to
# multiple variables!
a, b, c = x, y, z  # a=x; b=y; c=z

# The same process can be used to swap values!
a, b = b, a
```

This is mostly a useful shortcut (*syntactical sugar*), but is also used by some idiomatic Python constructions.

Resources

- Lists (Severance)
- Lists (Sweigart)
- Tuples (Downey)
- Sequence Types (Python Docs)

Chapter 9

Dictionaries

This chapter covers the second fundamental data structure in Python: **dictionaries**, which represent a collection of *key-value pairs*. They are similar to lists, except that each element in the dictionary is also given a distinct "name" to refer to it by (instead of an index number). Dictionaries are Python's primary version of **maps**, which is a common and extremely useful way of organizing data in a computer program—indeed, I would argue that maps are the *most useful* data structure in programming. This chapter will describe how to create, access, and utilize dictionaries to organize and structure data.

9.1   What is a Dictionary?

A **dictionary** is a lot like a *list*, in that it is a (one-dimensional) sequence of values that are all stored in a single variable. However, rather than using *integers* as the indexes for each of the elements, a dictionary allows you to use a wide variety of different data types (including *strings* and *tuples*) as the "index". These "indices" are called **keys**, and each is used to refer to a specific **value** in the collection. Thus a dictionary is an sequence of **key-value pairs**: each element has a "key" that is used to *look up* (reference) the "value".

This is a lot like a real-world dictionary or encyclopedia, in which the words (keys) are used to look up the definitions (values). A phone book works the same way (the names are the keys, the phone numbers are the values),

Dictionaries provide a **mapping** of keys to values: they specify a set of data (the keys), and how that data "transforms" into another set of data (the values).

Dictionaries are written as literals inside *curly braces* (**{}**). Key-value pairs are written with a *colon* (**:**) between the key and the value, and each element (pair) in the dictionary is separated by a *comma* (**,**):

```python
# A dictionary of ages
ages = {'sarah': 42, 'amit': 35, 'zhang': 13}

# A dictionary of English words and their Spanish translation
english_to_spanish = {'one':'uno', 'two':'dos'}

# A dictionary of integers and their word representations
num_words = {1: 'one', 2: 'two', 3: 'three'}

# Like lists, dictionary values can be of different types
# including lists and other dictionaries!
type_examples = {'integer': 12, 'string': 'dog', 'list': [1,2,3]}

# Each dictionary KEY can also be a different type
type_names = {511: 'an int key', 'hello': 'a string key',  (1,2): 'a tuple key!'}

# Dictionaries can be empty (with no elements)
empty = {}
```

Style Requirement: Dictionary variables are often named as plurals, but can also be named after the mapping they performed (e.g., `english_to_spanish`).

Be careful not to name a dictionary `dict`, which is a reserved keyword (it's a function used to create dictionaries).

Dictionary *keys* can be of any hashable type (meaning the computer can consistently convert it into a number). In practice, this means that that keys are most commonly *strings*, *numbers*, or *tuples*. Dictionary *values*, on the other hand, can be of any type that you want!

Dictionary *keys* must be unique: because they are used to "look up" values, there has to be a single value associated with each key (this is called a one-to-one mapping in mathematics). But dictionary *values* can be duplicated: just like how two words may have the same definition in a real-world dictionary!

```python
double_key = {'a': 1, 'b': 2, 'b': 3}
print(double_key)  # {'a': 1, 'b': 3}

double_val = {'a': 1, 'b': 1, 'c': 1}
print(double_val)  # {'a': 1, 'b': 1, 'c': 1}
```

It is important to note that dictionaries are an **unordered** collection of key-value pairs! Because you reference a value by its *key* and not by its position (as you do in a list), the exact ordering of those elements doesn't matter—the interpreter just goes immediately to the value associated with the key. This also means that when you print out a dictionary, the order in which the elements are printed may not match the order in which you specified them in the literal (and in fact, may differ between script executions or across computers!)

```
dict_a = {'a':1, 'b;':2}
dict_b = {'b':2, 'a:'1}
dict_a == dict_b  # True, order doesn't matter
```

The above examples mostly use dictionaries as "lookup tables": they provide a way of "translating" from some set of keys to some set of values. However, dictionaries are also extremely useful for grouping together related data—for example, information about a specific person:

```
person = {'first_name': "Ada", 'job': "Programmer", 'salary': 78000, 'in_union': True}
```

Using a dictionary allows you to track the different values with named keys, rather than needing to remember whether the person's name or title was the first element!

Dictionaries can also be created from *lists* of keys and values. To do this, you first use the built-in `zip()` function to create a non-list collection of tuples (each a key-value pair), and then use the built-in `dict()` function to create a dictionary out of that collection. Alternatively, the built-in `enumerate()` function will create an collection with the index of each list element as its key.

```
keys = ['key0', 'key1', 'key2']
values = ['val0', 'val1', 'val2']
dict(zip(keys, values))  # {'key0': 'val0', 'key1': 'val1', 'key2': 'val2'}
dict(enumerate(values))  # {0: 'val0', 1: 'val1', 2: 'val2'}
```

9.2   Accessing a Dictionary

Just as with lists, you retrieve a **value** from a dictionary using **bracket notation**, but you put the **key** inside the brackets instead of the positional index (since dictionaries are unordered!).

```
# A dictionary of ages
ages = {'sarah': 42, 'amit': 35, 'zhang': 13}

# Get the value for the 'amit' key
amit_age = ages['amit']
print(amit_age)  # 35

# Get the value for the 'zhang' key
zhang_age = ages['zhang']
print(zhang_age)  # 13

# Accessing a key not in the dictionary will give an error
print(ages['anonymous'])  # KeyError!
```

```python
# Trying to look up by a VALUE will give an error (since it's not a key)
print(ages[42])  # KeyError!
```

To reiterate: you put the *key* inside the brackets in order to access the *value*. You cannot directly put in a value in order to determine its key (because it may have more than one!)

It is worth noting that "looking up" a value by its key is a very "fast" operation (it doesn't take the interpreter a lot of time or effort). But looking up the key for a value takes time: you need to check each and every key in the dictionary to see if it has the value you're interested in! This concept is discussed in more detail in Chapter 11.

As with lists, you can put any *expression* (including variables) inside the brackets as long as it resolves to a valid key (whether that key is a string, integer, or tuple).

As with lists, you can **mutate** (change) the dictionary by assigning values to the bracket-notation variable. This changes the *key-value pair* to have a different value, but the same key:

```python
person = {'name': "Ada", 'job': "Programmer", 'salary': 78000}

# Assign a new value to the 'job' key
person['job'] = 'Sr Programmer'
print(person['job'])  # Sr Programmer

# Assign value to itself
person['salary'] = person['salary'] * 1.15  # a 15% raise!

# Add a new key-value pair by assigning a value to a key that is not yet
# in the dictionary
person['age'] = 37
print(person)  # {'name': 'Ada', 'job': 'Sr Programmer', 'salary': 89700.0, 'age': 3
```

Note that adding new elements (key-value pairs) works differently than lists: with a list, you cannot assign a value to an index that is out of bounds: you need to use the `append()` method instead. With a dictionary, you *can* assign a value to a non-existent key. This *creates* the key, assigning it the given value.

9.3 Dictionary Methods

Dictionaries support a few different operations and methods, though not as many as lists. These include:

```python
# A sample dictionary to demonstrate with
sample_dict = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}

# The standard `in` operator checks for operands in the keys, not the values!
'b' in sample_dict  # True, dict contains a key `'b'`
2 in sample_dict  # False, dict does not contain a key `2`

# The get() method returns the value for the key, or a "default" value
# if the key is not in the dictionary
default = -511  # a default value
sample_dict.get('c', default)  # 3, key is in dict
sample_dict.get('f', default)  # -511, key not in dict, so return default

# Remove a key-value pair
sample_dict.pop('d')  # removes and returns the `d` key and its value

# Replace values from one dictionary with those from another
other_dict = {'a':10, 'c': 10, 'n':10}
sample_dict.update(other_dict)  # assign values from other to sample
print(sample_dict)  # {'a': 10, 'b': 2, 'c': 10, 'e': 5, 'n': 10}

sample_dict.update(a=100, c=100)  # update also supports named arguments
print(sample_dict)  # {'a': 100, 'b': 2, 'c': 100, 'e': 5, 'n': 10}

# Remove all the elements
sample_dict.clear()
```

Dictionaries also include three methods that return list-like *sequences* of the dictionary's elements:

```python
sample_dict = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}

# Get a "list" of the keys
sample_keys = sample_dict.keys()
print(list(sample_keys))  # ['a', 'b', 'c', 'd', 'e']

# Get a "list" of the values
sample_vals = sample_dict.values()
print(list(sample_vals))  # [1, 2, 3, 4, 5]

# Get a "list" of the key-value pairs
sample_items = sample_dict.items()
print(list(sample_items))  # [('a', 1), ('b', 2), ('c', 3), ('d', 4), ('e', 5)]
```

The keys(), values(), and items() sequences are not quite lists (they don't have all of the list operations and methods), but they do support the in operator

and iteration with `for` loops (see below). And as demonstrated above, they can easily be converted *into* lists if needed. Note that the `items()` method produces a sequence of *tuples*—each key-value pair is represented as a tuple whose first element is the key and second is the value!

9.4   Dictionaries and Loops

Dictionaries are iterable collections (like lists, ranges, strings, files, etc), and so you can loop through them with a **for loop**. Note that the basic `for ...` `in ...` syntax iterates through the dictionary's *keys* (not its values)! Thus it is much more common to iterate through one of the `keys()`, `values()`, or `items()` sequences.

```python
sample_dict = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}

# Loop through the keys (implicitly)
for key in sample_dict:
    print(key, "maps to", sample_dict[key])  # e.g., "'a' maps to 1"

# Loop through the keys (explicitly)
for key in sample_dict.keys():
    print(key, "maps to", sample_dict[key])  # e.g., "'a' maps to 1"

# Loop through the values. Cannot directly get the key from this
for value in sample_dict.values():
    print("someone maps to", value)  # e.g., "someone maps to 1"

# Loop through the items (each is a tuple)
for item in sample_dict.items():
    print(item[0], "maps to", item[1])  # e.g., "'a' maps to 1"
```

It is *much* more common to use **multiple assignment** to give the `items()` tuple elements local variable names, allowing you to refer to those elements by name rather than by index:

```python
# Use this format instead!
for key, value in sample_dict.items():  # implicit  `key, value = item`
    print(key, "maps to", value)  # e.g., "'a' maps to 1

# Better yet, name the local variables after their semantic meaning!
for letter, number in sample_dict.items():
    print(letter, "maps to", number)  # e.g., "'a' maps to 1
```

Finally, remember that dictionaries are *unordered*. This means that there is no consistency as to which element will be processed in what order: you might get a then b then c, but you might get c then a then b! If the order is important

for looping, a common strategy is to iterate through a *sorted list of the keys* (produced with the built-in `sorted()` function):

```python
# Sort the keys
sorted_keys = sorted(sample_dict.keys())

# Iterate through the sorted keys
for key in sorted_keys:
    print(key, "maps to", sample_dict[key])

# Or all in one line!
for key in sorted(sample_dict.keys()):
    print(key, "maps to", sample_dict[key])
```

9.5   Nesting Dictionaries

Although dictionary *keys* are limited to hashable types (e.g., strings, numbers, tuples), dictionary *values* can be of any type—and this includes lists and other dictionaries!

Nested dictionaries are conceptually similar to nested lists, and are used in a similar manner:

```python
# a dictionary representing a person (spacing is for readability)
person = {
  'first_name': 'Alice',
  'last_name': 'Smith',
  'age': 40,
  'pets': ['rover', 'fluffy', 'mittens'],  # value is an array
  'favorites': {  # value is another dictionary
    'music': 'jazz',
    'food': 'pizza',
    'numbers': [12, 42]  # value is an array
  }
}

# Can assign lists or dicts to a new key
person['luggage_combo'] = [1,2,3,4,5]

# person['favorite'] is an (anonymous) dictionary, so can get that dict's 'food'
favorite_food = person['favorites']['food'];

# Get to the (anonymous) 'favorites' dictionary in person, and from that get
# the (anonymous) 'numbers' list, and from that get the 0th element
first_fav_number = person['favorite']['numbers'][0];  # 12
```

```python
# Since person['favorite']['numbers'] is a list, we can add to it
person['favorite']['numbers'].append(7);  # add 7 to end of the list
```

The ability to nest dictionaries inside of dictionaries is incredibly powerful, and allows you to define arbitrarily complex information structurings (schemas). Indeed, most data in computer programs—as well as public information available on the web—is structured as a set of nested maps like this (though possibly with some level of abstraction).

The other common format used with nested lists and dictionaries is to define a **list of dictionaries** where each dictionary has *the same keys* (but different values). For example:

```python
# Arbitrary list of people's names, heights, and weights
people = [
    {'name': 'Ada', 'height': 64, 'weight': 135},
    {'name': 'Bob', 'height': 74, 'weight': 156},
    {'name': 'Chris', 'height': 69, 'weight': 139},
    {'name': 'Diya', 'height': 69, 'weight': 144},
    {'name': 'Emma', 'height': 71, 'weight': 152}
]
```

This structure can be seen as a list of **records** (the dictionaries), each of which have a number of different **features** (the key-value pairs). This list of feature records is in fact a common way of understanding a **data table** like you would create as an Excel spreadsheet:

name	height	weight
Ada	64	135
Bob	74	156
Chris	69	139
Diya	69	144
Emma	71	152

Each dictionary (record) acts as a "row" in the table, and each key (feature) acts as a "column". As long as all of the dictionaries share the same keys, this list of dictionaries *is* a table!

When working with large amounts of tabular data, like you might read from a `.csv` file, this is a good structure to use.

In order to analyze this kind of data table, you most often will loop through the elements in the list (the rows in the table), doing some calculations based on *each* dictionary:

```
# How many people are taller than 70 inches?
taller_than_70 = 0
for person in people:  # iterate through the list
    # person is a dictionary
    if person['height'] >= 70:  # each dictionary has a 'height' key
        taller_than_70 += 1  # increment the count

print(taller_than_70)  # 2
```

This is effective, but not particularly efficient (or simple). We will discuss more robust and powerful ways of working with this style of data table more in a later chapter.

9.6 Which Data Structure Do I Use?

The last two chapters have introduced multiple different **data structures** built into Python: *lists*, *tuples*, and *dictionaries*. They all represent collections of elements, though in somewhat different ways. So when do you each each type?

- Use **lists** for any *ordered* sequence of data (e.g., if you care about what comes first), or if you are collecting elements of the same general "type" (e.g., a lot of numbers, a lot of strings, etc.). If you're not sure what else to use, a *list* is a great default data structure.

- Use **tuples** when you need to ensure that a list is *immutable* (cannot be changed), such as if you want it to be a key to a dictionary or a parameter to a function. Tuples are also nice if you just want to store a *small* set of data (only a few items) that is not going to change. Finally, tuples may potentially provide an "easier" syntax than lists in certain situations, such as when using multiple assignment.

- Use **dictionaries** whenever you need to represent a *mapping* of data and want to link some set of keys to some set of values. If you want to be able to "name" each value in your collection, you can use a dictionary. If you want to work with key-value pairs, you need to use a dictionary (or some other dictionary-like data structure).

Resources

- Dictionaries and Data Structures (Sweigart)
- Dictionaries (Downey)

Chapter 10

Searching and Efficiency

One of the most common and important uses of computer programs is to **search** a large set of data for a specific record or observation—indeed, a significant portion of what computers *do* is search through data. This chapter covers a number of common *patterns* and *algorithms* used when searching through lists of data in order to answer questions about that data. Note that this chapter introduces no new syntax, but instead provides a deeper look at using *loops* and *lists*. The first section illustrates common interactions with lists, while the remaining consider at a high level the efficiency of algorithms.

10.1   Linear Search

Fundamentally, search algorithms are used to "find" a particular item in list: given a very large list of elements, the goal of the search is to determine *whether* the list contains the "target" item, and if so *where* in the list that element is. Thus basic search algorithms are used to answer the questions ***is an item in a list?*** or ***which element in the list is the item?***.

Python does contain built-in operators and list methods (e.g., `in`, `index()`, `max()` etc.) that can answer simple versions of these questions. However, more complex programs may require you to create your own "custom" searches following the patterns described here.

The most basic algorithm you can use to answer these questions is called a **linear search**. Intuitively, this search takes all of the elements in a list, and then goes down the "line" of elements one after another, *checking* if each element in turn is the target item. (Think: "Are you who I'm looking for?" "No" "Are you who I'm looking for?" "No", "Are you who I'm looking for?" "Yes!"). If you get through the entire line of items *without* finding the target (without anyone

answering "yes"), then you know that the target is not in the list (because you checked everyone)!

This kind of search involves just a simple `for` loop (to consider every element) and `if` statement (to check if the item is the target):

```python
def linear_in(a_list, target):
    """Searches the given list for the given target value.
       Returns whether or not the target is in the list"""

    for element in a_list:      # go through each element
        if element == target:  # check that element
            return True        # if found, report so!

    return False  # looked at everyone but didn't find, report back

# Example:
numbers = [17, 18, 3, 7, 11, 16, 13, 4]  # the list to search
print( linear_in(numbers, 11) )  # True, 11 in list
print( linear_in(numbers, 12) )  # False, 12 not in list
```

Pay careful attention to the position of the `return` statements! First, you return `True` if (when) the target is found—this "exits" the function so you do not keep searching once you've found your target. The second `return` statement occurs *after the loop has entirely finished*. You need to check *every* item in the list before you can conclusively say that the item isn't there (otherwise you may have just missed it).

A common error is include an `else` clause that returns or stores that the item was not found. However, returning `False` makes the statement "**none** of the elements in the list is the target" which is dependent on the entire list: looking at a single item won't let you make that claim! Contrast this with returning `True`, which makes the statement "**one** of the elements in the list is the target", which can be proved by considering just one item (e.g., the *one*).

While a useful organizing tool, a linear search does not need to be implemented as its own function if you instead use a variable to track whether the item is found or not:

```python
found = False  # has not been found when we start looking
for element in a_list:   # go through each element
  if element == target:  # check that element
      found = True       # mark as found!
```

To determine **which** element in the list is the target, you use the same structure but consider the *index* of each element, returning that index when the item is found. By convention, if the target is *not* in the list, you return `-1` (which will be an out-of-bounds index for any list):

```python
def linear_search(a_list, target):
    """Searches the given list for the given target value.
       Returns the index of the target, or -1 if target not in the list"""

    for index in range(len(a_list)):  # go through each element
        if a_list[index] == target:  # check that element
            return index  # if found, report the index

    return -1  # looked at everyone but didn't find, report back

# Example:
numbers = [17, 18, 3, 7, 11, 16, 13, 4] # the list to search
print( linear_search(numbers, 11) )  # 4
print( linear_search(numbers, 12) )  # -1
print( linear_search(numbers, 11) >= 0 )  # True, 11 in list
print( linear_search(numbers, 12) >= 0 )  # False, 12 not in list
```

10.1.1 Maximal Search

Another common item to search for is the "biggest" (or "smallest") element in the list. This may be the biggest number, the longest word, the slowest turtle, the highest-scoring sports game... any element that has a "greater" ordinal value (e.g., it comes "first" in some ordered listing, whether that ordering is *ascending* or *descending*). Whenever you are searching for the "-est" item in a list, you can use the same variant on a linear search.

I refer to this variant as a "king-of-the-hill" search, named after the children's game. In this algorithm, start be declaring an initial element (often the first in the list) as the "king"—the "greatest" value in the list. The algorithm then goes down the "line" of elements, having each one in turn "challenge" the king. If the challenging value is greater, then that value becomes the new king, and the process continues. Whichever item is the king in the end must be the "greatest" item that was being searched for.

This algorithm is implemented with a similar structure to the basic linear search, except instead of comparing to the target, you compare to the current "king":

```python
def maximum(a_list):
    """Returns the element with the maximum value in the list."""

    maximum = a_list[0]  # first person starts as the "king"
    for element in a_list:  # go through each element
        if element > maximum:  # challenge the king
            maximum = element  # if won, become the new king
```

```
    return maximum  # in the end, return who is left standing

# Example:
numbers = [17, 18, 3, 7, 11, 16, 13, 4] # the list to search
print( maximum(numbers) )  # 18
```

The most common error with this algorithm is not comparing to the *previous maximum*, instead trying to compare to e.g, the previous element in the list.

Note that it is also possible to do this same search by using the built-in `max()` function and specifying an *ordering function* as an argument, which converts any element into a value with the proper ordering. Using functions as arguments is discussed in the next chapter.

To reiterate, this function can be used to find any extreme value simply by changing the "challenge" comparison. For example use < instead of > to find the "minimum" item, or use a more complex boolean expression to compare dictionary "rows" in a data table.

10.1.2   Falsification Search

Many searches are interested in determining if *all* elements in a list meet a certain criteria. For example, determining if all the numbers are greater than 10, if all of the words are less than 3 syllables, if all of the turtles are running at speed, or if the home team won all of their games.

But loops are only able to consider one element at a time, not "all" the elements at once. So in order to answer these questions, you need to **invert** the question: saying "*all* numbers are *greater* than 10" is logically equivalent to saying "*no* number is *less* than (or equal to) 10".

(Somewhat counter-intuitively, the logical negation of an *"all are"* predicate is not *"none are"* but *"**one is not**"*). That is, the opposite of "all days are sunny" is not "no days are sunny" but "(at least) one day was not sunny".

By inverting the question into a search for a **counter-example**, you can utilize the previous linear search pattern:

```
def all_larger(a_list, minimum):
    """Returns whether all of the elements in the list are larger than the
       given minimum value."""

    all_large = True  # every number we've looked at is large
    for element in a_list:
        if not (element > minimum):  # written as counter-example
                                     # equivalent to `element <= minimum`
            all_large = False  # counterexample found! Statement no longer valid
```

```
        break  # "exits" from the loop, since don't need to search more

    return all_large  # report back

# Example:
numbers = [17, 18, 3, 7, 11, 16, 13, 4] # the list to search
print( all_larger(numbers, 10) )  # False, some numbers are smaller than 10
print( all_larger(numbers, 2) )  # True, all numbers are greater than 2
```

Careful variable naming (e.g., `all_large`) for keeping track of boolean claims is vital to being able to read and write these algorithms!

Overall, a linear search is a simple and versatile algorithm, but can be tricky to apply depending on the question being asked.

10.2   Linear Search Speed

Searching is something that you do a lot (and on bigger and bigger data sets!), so it's worth considering: how fast is this process? How *efficient* is the algorithm? Is there possibly a more efficient way of searching?

Warning: *AVOID PRE-MATURE OPTIMIZATION*!! While this section and chapter discuss the "speed" of computer programs, you should avoid spending too much (or any!) time trying to make your program "as fast as possible". The first step in any computer program is to make it function at all. Once you have it working, *then* you can worry about increasing the efficiency—and only if it is currently too slow for your purposes. Pre-mature optimization (trying to make it work fast before you make it work at all) is a major source of bugs and other problems.

One way of measuring the speed of an algorithm would be to *time it*, such as by using a stopwatch. Python includes modules that can be used to record the time, allowing you to get the "start" and "stop" time of the algorithm, and then calculate the elapsed duration. However, this **wall-clock efficiency** is highly dependent on the exact list being searched and on the computer that is executing the algorithm—if you're also streaming videos while searching, the algorithm may run slower!

Instead, computer scientists measure the efficiency of a program by counting the number of *operations* that the algorithm does. This number will be independent of the data and machine, and so makes it easier to compare approaches. Specifically, to measure the efficiency of a search, you would consider the number of **comparisons** that need to be made between elements (e.g., how many elements you check before you find what you're looking for), under the assumption that this is the most time-consuming part of the computer's search algorithm.

Linear search involves looking at each item in the list one at a time, so the number of "checks" that need to be made is *dependent on the size of the list.* And because the item you're looking for may be either be at the beginning of the list (meaning you don't search for very long) or at the end, you can consider both the "average" case (when it's in the middle), as well as the "worst" case (when it's at the end or not in the list at all!):

len(list)	# comparisons (avg)	# comparisons (worst)
10	5	10
20	10	20
50	25	50
100	50	100
1000	500	1000
N	**N/2**	**N**

These numbers should be somewhat intuitive: because you're looking at each element in a line, in the average case you need to look at half of the elements, and in the worst case you need to look at all of them! As such, the number of comparisons you need to make (and thus the efficiency of the algorithm) is a **linear function** of the size of the list:
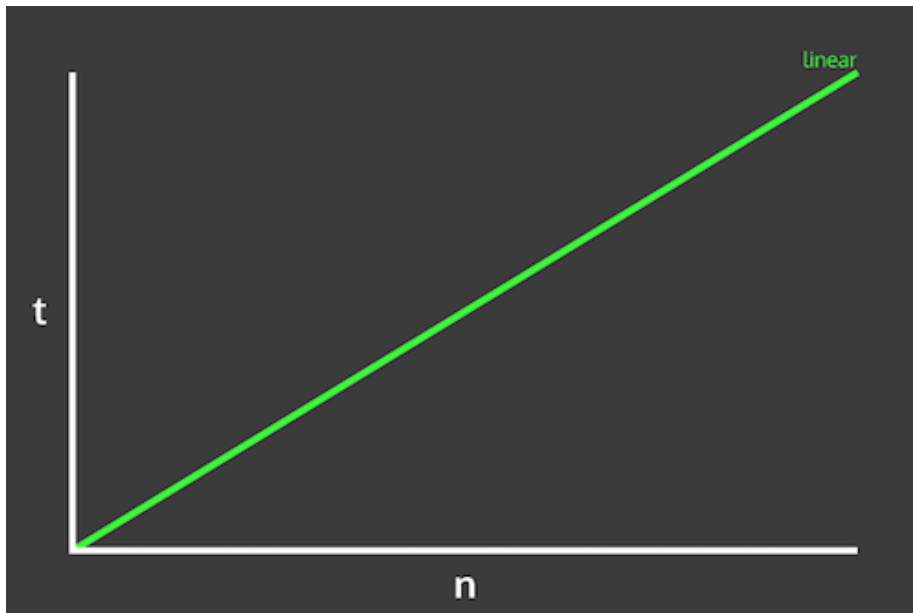
Figure 10.1: Linear time complexity. Image by Nick Salloum.

This is in fact why it is called a ***linear*** search!

In general, we measure algorithm efficiency (or more properly, **algorithmic complexity**) in terms of the *rate of change* in the speed: that is, if you double the size of the input list, by what ratio does the work you need to do increase? With a linear search, doubling the size of the list will double the amount of work to do.

Note that looking up an element by its index or key is a **constant function**—it takes the same amount of time no matter how big the list or dictionary is. This is part of why dictionaries are so useful as look-up tables: you don't need to spend time searching for the value if it you have its key!

10.3 Binary Search

There are alternate, faster algorithms for searching through lists that can be useful as the data set gets large.

As an example, consider how you might search for a name in a phone book (or a word in an encyclopedia): rather than starting from "A" and going one by one though the book, you flip it open to the middle. If the name you're after comes later in the alphabet than the page you opened to, then you take the "back half" of the book and flip to the middle of that (otherwise, you flip to the middle of the "front half"). You repeat this process, narrowing the number of pages you need to consider more and more until you've found the name you are looking for!

What makes this work is the fact that names in the phone book are **ordered**: after doing a comparison (checking the name against the page), you know whether to go forward or backwards to find the target.

This algorithm is known as a **binary search**, and it lets you search an *ordered* list by comparing to the middle, and reducing the list to only the top or bottom half, and repeating:

```python
def binary_search(a_list, target):
    """Searches the given SORTED list for the given target value.
       Returns the index of the target, or -1 if target not in the list"""

    start_index = 0 # initial goalposts
    end_index = len(a_list)-1

    while start_index <= end_index:  # at least one thing to look at
        middle_index = (start_index + end_index) // 2  # middle (integer) index

        if(a_list[middle_index] == target):
            return middle_index  # found the item!
```

```python
        elif target > a_list[middle_index]:
            start_index = middle_index+1  # move goalpost
        else:
            end_index = middle_index-1  # move goalpost

    return -1  # did not find the item
```

This algorithm starts by considering the entire list (with the "goalposts" at either end. It then looks at the middle element. If that isn't the target, then it moves the appropriate goalpost to that middle spot, thereby throwing away the half of the list and narrowing the search field. This continues until the target is found or the goalposts have moved "past" one another, at which point the search ends. See this animation for an example of how it works.

(Note that this is an example of *indefinite iteration*: you don't know how many times you'll need to cut the list in half before searching, so you use a `while` loop!)

10.3.1   Binary Search Speed

Using a timer will demonstrate that *binary search* is much faster than *linear search* (as the list gets large), but how much faster?

The intuition behind the speed of a binary search is as follows: Each time through the loop (each "comparison") effectively lets you reduce the size of the list by *half*. So if there are `N` items in the list, the first time through the loop reduces the list to `N*(1/2)` items, the second time through reduces is to `N*(1/2)*(1/2)`, etc. In the worst case, you will need to cut the list in half until you have reduced the list to exactly one item, or:

$$N * \frac{1}{2^{loops}} = 1$$

We can solve this equation for the number of `loops`:

$$N = 2^{loops}$$

$$loops = log_2(N)$$

Thus binary search's speed is a **logarithmic function** of the size of the list: that is, if you double the size of the list, the number of comparisons you need to do (the number of times through the loop) *increases by just 1.*

This is *drastically* faster than a linear search…. however, it requires a list be sorted for it to work!
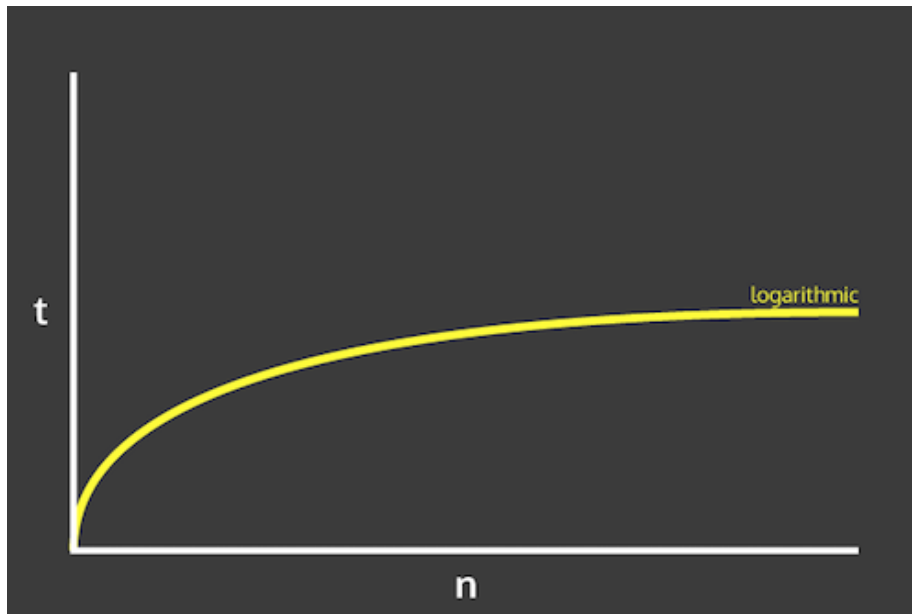
Figure 10.2: Logarithmic time complexity. Image by Nick Salloum.

10.4 Sorting

There are many, *many* different algorithms for sorting numbers, many of which have amusing names given to them by computer scientists (Python's built-in `sorted()` method uses one called Timsort, named after the man who invented it). This chapter will discuss just one straightforward example for illustration purposes.

One such algorithm (**Selection Sort**) utilizes the "king-of-the-hill" search described above. This algorithm works as follows: search for ("select") the smallest item in the list. Because it is the smallest, it must be the first item in the sorted list, and can be placed there. Next, select the second-smallest item in the list (the smallest of the "unsorted" items), and place that second in the "sorted" list. Continue with this process until the entire list is sorted!

```python
def selection_sort(a_list):
    """Sorts the list (in place)"""

    for i in range(len(a_list)):  # go through each spot in the list
        # Do a "king-of-the-hill" search of the remaining items
        selected_index = i
        for j in range(i, len(a_list)):
```

```
        if(a_list[j] < a_list[selected_index]):
            selected_index = j

    # swap smallest into place (multi-assignment!)
    a_list[i], a_list[selected_index] = a_list[selected_index], a_list[i]
```

To determine the speed of the *selection sort* algorithm, notice that the first time through the loop requires considering `N` different items (the whole loop). The second time requires checking `N-1` items, the third time `N-2` items, and so forth until the last time through the loop you only need to compare `2` then `1` items. These checks can then be summed into a series:

$$N + (N - 1) + (N - 2) + ... + 2 + 1$$
$$= \frac{N(N+1)}{2}$$
$$\approx N^2$$

Selection sort's speed is a **quadratic function** of the size of the list: that is, if you double the size of the list, the number of comparisons you need to do *quadruples!*.

Faster sorting algorithms (like Python's Timsort) get this speed down to "fast" `N*log2(N)` ("loglinear"), which is much better than *quadratic* algorithms but still notably slower than *linear* algorithms.

For large lists, sorting a list is slower than just using a linear search on it… but once that list is sorted, you can use the ultra-fast binary search! This is a **tradeoff** that needs to be considered when trying to improve the efficiency of programs that work on large data sets: sometimes you need to spend some extra time up front to prepare (sort) the data, in order to be able to utilize (search) it more effectively.

Resources

- List Algorithms (Downey), Sections 14.1 - 14.7
- What is Algorithm Analysis? and Big-O Notation

Chapter 11

Functional Programming

This chapter introduces techniques from **Functional Programming**, which is a programming paradigm centered on *functions* rather than on *variables* and statements as you've been doing so far (known as *imperative programming*). Functional programming offers another way to think about giving "instructions" to a computer, which can make it easier to think about and implement some algorithms. While not a completely functional language, Python does contain a number of "functional-programming-like" features that can be mixed with the imperative strategies we're used to, allowing for more compact, readable, and "Pythonic" code in some cases.

11.1   Functions ARE Variables

Previously we've described functions as "named sequences of instructions", or groupings of lines of code that are given a name. But in a functional programming paradigm, functions are *first-class objects*—that is, they are "things" (values) that can be organized and manipulated *just like variables*.

In Python, **functions ARE variables**:

```python
# create a function called `say_hello`
def say_hello(name):
    print("Hello, "+name)

# what kind of thing is `say_hello` ?
type(say_hello)  # <class 'function'>
```

Just like x = 3 defines a variable for a value of type int, or msg = "hello" defines a variable for a value of type string, the above say_hello function is actually a variable for a *value* of type function!

Note that we refer to the function by its name *without* the parentheses!

This is why it is possible to accidentally "overwrite" built-in functions by assigning values to variables like `sum`, `max`, or `dict`.

The fact that functions **are** variables is the core realization to make when programming in a functional style. You need to be able to think about functions as **things** (nouns), rather than as **behaviors** (verbs). If you imagine that functions are "recipes", then you need to think about them as *pages from the cookbook* (that can be bound together or handed to a friend), rather than just the sequence of actions that they tell you to perform.

And because functions are just another type of variable, they can be used **anywhere** that a "regular" variable can be used. For example, functions are values, so they can be assigned to other variables!

```python
# Create a function `say_hello`
def say_hello(name):
    print("Hello, "+name)

# Assign the `say_hello` value to a new variable `greet`
greet = say_hello

# Call the function assigned to the `greet` variable
greet("world")  # prints "Hello world"
```

It helps to think of functions as just a special kind of list. Just as *lists* have a special syntax [] (bracket notation) that can be used to "get" a value from the list, *functions* have a special syntax () (parentheses) that can be used to "call" and run the function.

Moreover, functions are values, so they can be *passed as parameters to other functions*!

```python
# Create a function `say_hello`
def say_hello(name):
    print("Hello, "+name)

# A function that takes ANOTHER FUNCTION as an argument
# This function will call the argument function, passing it "world"
def do_with_world(func_to_call):
  # call the given function with an argument of "world"
  func_to_call("world")

# Call `do_with_world`, saying the "thing to do" is `say_hello`
do_with_world(say_hello)  # prints "Hello world"
```

In this case, the `do_with_world` function will *call* whatever function it is given, passing in a value of `"world"`. (You can think of this as similar to having a

function that accesses the `'world'` key of a given dictionary).

Important note: when you pass `say_hello` as an argument, you don't put any parentheses after it! Putting the parentheses after the function name *calls* the function, causing it to perform the lines of code it defines. This will cause the expression containing the function to *resolve* to its returned value, rather than being the function value itself. It's like passing in the baked cake rather than the recipe page.

```python
def greet():  # version with no args
    return "Hello"

# Print out the function
print(greet)  # prints <function greet>, the function

# Resolve the expression, then print that out
print(greet())  # prints "Hello", which is what `greet()` resolves to.
```

A function that is passed into another is commonly referred to as a **callback function**: it is an argument that the other function will "call back to" and execute when needed.

Functions can take more than one *callback function* as arguments, which can be a useful way of *composing* behaviors.

```python
def do_at_once(first_callback, second_callback):
    first_callback()  # execute the first function
    second_callback()  # execute the second function
    print("at the same time! ")

def pat_head():
    print("pat your head", end=" ")

def rub_belly():
    print("rub your belly", end=" ")

# Pass in the callbacks to "do at once"
do_at_once(pat_head, rub_belly)
```

This idea of *passing functions are arguments to other functions* is at the heart of functional programming, and is what gives it expressive power: you can define program behavior primarily in terms of the behaviors that are run, and less in terms of the data variables used.

11.1.1   lambdas: Anonymous Functions

You have previously used **anonymous variables** in your programs, or values which are not assigned a variable name (so remain anonymous). These values were defined as *literals* or expressions and passed directly into functions, rather than assigning them to variables:

```python
my_list = [1,2,3]  # a named variable (not anonymous)
print(my_list)  # pass in non-anonymous variable
print([1,2,3])  # pass in anonymous value
```

Because functions **are** variables, it is also possible to define **anonymous functions**: functions that are not given a name, but instead are passed directly into other functions. In Python, these anonymous functions are referred to as **lambdas** (named after lambda calculus, which is a way of defining algorithms in terms of functions).

Lambdas are written using the following general syntax:

```python
lambda arg1, arg2: expression_to_return
```

You indicate that you are defining a lambda function with the keyword `lambda` (rather than the keyword `def` used for named functions). This is followed by a list of arguments separated by commas (what normally goes inside the `()` parentheses in a named function definition), then a colon `:`, then the expression that will be *returned* by the anonymous function.

For example, compare the following named and anonymous function definitions:

```python
# Named function to square a value
def square(x):
    return x**2

# Anonymous function to square a value
lambda x: x**2

# Named function to combine first and last name
def make_full_name(first, last):
    return first + " " + last

# Anonymous function to combine first and last name
lambda first, last: first + " " + last
```

You're basically replacing `def` and the function name with the word `lambda`, removing the parentheses around the arguments, and removing the `return` keyword!

Just as other expressions can be assigned to variables, lambda functions can be assigned to variables in order to give them a name. This is the equivalent of

having defined them as named functions in the first place:

```python
square = lambda x: x**2
make_full_name = lambda first, last: first + " " + last
```

There is one major restriction on what kind of functions can be defined as anonymous lambdas: they must be functions that consist of **only** a single returned expression. That is, they need to be a function that contains exactly one line of code, which is a `return` statement (as in the above examples). This means that lambdas are **short** functions that usually perform very simple transformations to the arguments... exactly what you want to do with functional programming!

11.2 Functional Looping

Why should you care about treating functions as variables, or defining anonymous lambda functions? Because doing so allows you to *replace loops with function calls* in some situations. For particular kinds of loops, this can make the code more *expressive* (more clearly indicative of what it is doing).

11.2.1 Map

For example, consider the following loop:

```python
def square(n):  # a function that squares a number
    return n**2

numbers = [1,2,3,4,5]  # an initial list

squares = []  # the transformed list
for number in numbers:
    transformed = square(number)
    squares.append(transformed)
print(squares)  # [1, 4, 9, 16, 25]
```

This loop represents a **mapping** operation: it takes an original list (e.g., of numbers 1 to 5) and produces a *new* list with each of the original elements transformed in a certain way (e.g., squared). This is a common operation to apply: maybe you want to "transform" a list so that all the values are rounded or lowercase, or you want to *map* a list of words to a list of their lengths. It is possible to make these changes uses the same pattern as above: create an empty list, then loop through the original list and `append` the transformed values to the new list.

However, Python also provides a *built-in function* called **map()** that directly performs this kind of mapping operation on a list without needing to use a

loop:

```python
def square(n):  # a function that squares a number
    return n**2

numbers = [1,2,3,4,5]  # an initial list

squares = list(map(square, numbers))
print(squares)  # [1, 4, 9, 16, 25]
```

The `map()` function takes a list and produces a *new* list with each of the elements transformed. The `map()` function should be passed two arguments: the second is the list to transform, and the first is the *name of a callback function* that will do the transformation. This callback function must take in a *single* argument (an element to transform) and return a value (the transformed element).

Note that in Python 3, the `map()` function returns an *iterator*, which is a list-like sequence similar to that returned by a dictionary's `keys()` or `items()` methods. Thus in order to interact with it as a list, it needs to be converted using the `list()` function.

The `map()` callback function (e.g., `square()` in the above example) can also be specified using an anonymous lambda, which allows for concisely written code (but often at the expense of readability—see *List Comprehensions* below for a more elegant, Pythonic solution).

```python
numbers = [1,2,3,4,5]  # an initial list
squares = list(map(lambda n:n**2, numbers))
```

11.2.2   Filter

A second common operation is to **filter** a list of elements, removing elements that you don't want (or more accurately: only keeping elements that you DO want). For example, consider the following loop:

```python
def is_even(n):  # a function that determines if a number is even
    remainder = n % 2  # get remainder when dividing by 2 (modulo operator)
    return remainder == 0  # True if no remainder, False otherwise

numbers = [2,7,1,8,3]  # an initial list

evens = []  # the filtered list
for number in numbers:
    if is_even(number):
        evens.append(number)
print(evens)  # [2, 8]
```

With this **filtering** loop, we are *keeping* the values for which the `is_even()` function returns true (the function determines "what to let in" not "what to keep out"), which we do by appending the "good" values to a new list.

Similar to `map()`, Python provides a *built-in function* called **filter()** that will directly perform this filtering:

```python
def is_even(n):  # a function that determines if a number is even
    return (n % 2) == 0  # True if no remainder, False otherwise

numbers = [2,7,1,8,3]  # an initial list

evens = list(filter(is_even, numbers))
print(evens)  # [2, 8]
```

The `filter()` function takes a list and produces a *new* list that contains only the elements that *do match* a specific criteria. The `filter()` function takes in two arguments: the second is the list to filter, and the first is the *name of a callback function* that will do the filtering. This callback function must take in a *single* argument (an element to consider) and return `True` if the element should be included in the filtered list (or `False` if it should not be included).

Because `map()` and `filter()` both produce list-like sequences, it is possible to take the returned value from one function and pass it in as the argument to the next. For example:

```python
numbers = [1,2,3,4,5,6]

# Get the squares of EVEN numbers only
filtered = filter(is_even, numbers)  # filter the numbers
squares = map(square, filtered)  # map the filtered values
print(list(squares))  # [4, 16, 36]

# Or in one statement, passing results anonymously
squares = map(square,
          filter(is_even,
          numbers))  # watch out for the parentheses!
print(list(squares))  # [4, 16, 36]
```

This structure can potentially make it easier to understand the code's intent: it is "squareing the `is_even` numbers"!

11.2.3 Reduce

The third important operation in functional programming (besides *mapping* and *filtering*) is **reducing** a list. Reducing a list means to *aggregate* that lists values together, transforming the list into a single value. For example, the built-in

`sum()` function is a *reducing* operation (and in fact, the most common one!): it reduces a list of numbers to a single summed value. Thus you can think of `reduce()` as a *generalization* of the `sum()` function—but rather than just adding (+) the values together, `reduce()` allows you to specify what operation to perform when aggregating (e.g., multiplication).

Because the `reduce()` function can be complex to interpret, it was actually *removed* from the set of "core" built-in functions in Python 3 and relegated to the `functools` module. Thus you need to import the function in order to use it:

```python
from functools import reduce
```

To understand how a *reduce* operation works, consider the following basic loop:

```python
def multiply(x, y): # a function that multiplies two numbers
    return x*y

numbers = [1,2,3,4,5]  # an initial list

running_total = 1  # an accumulated aggregate
for number in numbers:
    running_total = multiply(running_total, number)
print(running_total)  # 120  (1*2*3*4*5)
```

This loop **reduces** the list into an "accumulated" product (factorial) of all the numbers in the list. Inside the loop, the `multiply()` function is called and passed the "current total" and the "new value" to be combined into the aggregate (*in that order*). The resulting total is then reassigned as the "current total" for the next iteration.

The **reduce()** function does exactly this work: it takes as arguments a *callback* function used to combine the current running total with the new value, and a list of values to combine. Whereas the `map()` and `filter()` callback functions each took 1 argument, the `reduce()` callback function requires **2** arguments: the first will be the "running total", and the second will be the "new value" to mix into the aggregate. (While this ordering doesn't influence the factorial example, it is relevant for other operations):

```python
def multiply(x, y): # a function that multiplies two numbers
    return x * y

numbers = [1,2,3,4,5]  # an initial list

product = reduce(multiply, numbers)
print(product)  # 120
```

Note that the `reduce()` function aggregates into a single value, so the result doesn't need to be converted from an *iterator* to a list!

To summarize, the `map()`, `filter()`, and `reduce()` operations work as follows:

Figure 11.1: Map, filter, reduce explained with emoji.

All together, the **map**, **filter**, and **reduce** operations form the basis for a functional consideration of a program. Indeed, these kinds of operations are very common when discussing data manipulations: for example, the famous MapReduce model involves "mapping" each element through a complex function (on a different computer no less!), and then "reducing" the results into a single answer.

11.3   List Comprehensions

While `map()` and `filter()` are effective ways of producing new lists from old, they can be somewhat hard to read (particularly when using anonymous lambda functions, which you often would want to do for simple transformations). Instead, a more idiomatic and "Pythonic" approach (preferred by language developer Guido van Rossum) is to use **List Comprehensions**. A *list comprehension* is a special syntax for doing mapping and/or filtering operations on list using the `for` and `if` keywords you are familiar with.

A basic list comprehension has the following syntax:

```
new_list = [output_expression for loop_variable in sequence]
```

For example, a list comprehension to **map** from a list of numbers to their squares would be:

```
numbers = [1,2,3,4,5]  # original list
squares = [n**2 for n in numbers]
print(squares)  # [1, 4, 9, 16, 25]
```

List comprehensions are written inside square brackets `[]` and use the same `for ... in ...` syntax used in for loops. However, the *expression* that you would normally `append()` to the output list when mapping (or that is returned from an anonymous lambda function) is written *before* the `for`. This causes the above comprehension to be read as *"a list consisting of* `n**2` *(n squared) for each* `n` *in* `numbers`*"*—it's almost English!

You can contrast a list comprehension with the same mapping operation done via a loop or via a `map()` and a lambda:

```
# with a loop
squares = []
for n in numbers:
    squares.append(n**2)  # append expression

# with a lambda
squares = list(map(lambda n: n**2, numbers))  # map with lambda

# with a list comprehension
squares = [n**2 for n in numbers]  # map with list comprehension
```

Notice that all 3 versions specify a *transformation expression* (`n**2`) on a input variable (`n`). They just use different syntax (punctuation and ordering) to specify the transformation that should occur.

List comprehensions can also be used to **filter** values (even as they are being mapped). This is done by specifying an `if` filtering condition after the sequence:

```
new_list = [output_expression for loop_variable in sequence if condition]
```

Or as a specific example (remember: we filter for elements to *keep!*):

```
numbers = [2,7,1,8,3]
evens = [n for n in numbers if n%2 == 0]
print(evens)  # [2, 8]
```

This can be read as *"a list consisting of* `n` *for each* `n` *in* `numbers`*, but only if* `n%2 == 0`*"*. It is equivalent to using the `for` loop:

```
evens = []
for n in numbers
    if n%2 == 0:  # check the filter condition
        evens.append(n)  # append the expression
```

Finally, it is possible to include *multiple, nested* `for` and `if` statements in a list comprehension. Each successive `for ... in ...` or `if` expression is

included inside the square brackets after the output expression. This allows you to effectively convert nested control structures into a comprehension:

```python
entrees = ["chicken","fish","veggies"]
sides = ["potatoes", "veggies"]

# Get all "meals" if the entree and side are not the same
meals = [ entree+" & "+side for entree in entrees for side in sides if entree != side]
print(meals)  # ['chicken & potatoes', 'chicken & veggies', 'fish & potatoes',
              #  'fish & veggies', 'veggies & potatoes']
              # Note: no "veggies and veggies" !
```

This is equivalent to the nested loops.

```python
meals = []
for entree in entrees:
    for side in sides:
        if entree != side:
            meals.append(entree+" & "+side)
```

(This *almost* acts like a **reduce** operation, reducing two lists into a single one... but it doesn't exactly convert).

Overall, list comprehensions are considered a *better, more Pythonic* approach to functional programming. However, `map()` `filter()` and `reduce()` functions are a more generalized approach that can be found in multiple different languages and contexts, including other data-processing languages such as R, Julia, and JavaScript. Thus it is good to be at least familiar with both approaches!

Resources

- Functional Programming in Python (IBM) (note: Python 2)
- Map, Filter, Lambda, and List Comprehensions in Python (note: Python 2)
- Functional Programming in Python (O'Reilly) (short eBook)
- List Comprehensions (Python Docs)
- List Comprehensions Explained Visually
- Functional Programming HOWTO (advanced, not recommended)

Chapter 12

Accessing Web APIs

You've used Python to load data from external files (either text files or locally-saved `.csv` files), but it is also possible to programmatically download data directly from web sites on the internet. This allows scripts to always work with the latest data available, performing analysis on data that may be changing rapidly (such as from social networks or other live events). Web services may make their data easily accessible to computer programs like Python scripts by offering an **Application Programming Interface (API)**. A web service's API specifies *where* and *how* particular data may be accessed, and many web services follow a particular style known as *Representational State Transfer (REST)*. This chapter will cover how to access and work with data from these *RESTful APIs*.

12.1   Web APIs

An **interface** is the point at which two different systems meet and *communicate*: exchanging informations and instructions. An **Application Programming Interface (API)** thus represents a way of communicating with a computer application by writing a computer program (a set of formal instructions understandable by a machine). APIs commonly take the form of **functions** that can be called to give instructions to programs—the set of functions provided by a module like `math` or `turtle` make up the API for that module. While most APIs provide an interface for utilizing *functionality*, other APIs provide an interface for accessing *data*. One of the most common sources of these data apis are **web services**: websites that offer an interface for accessing their data.

- (Technically the interface is just the function **signature** which says how you *use* the function: what name it has, what arguments it takes, and what value it returns. The actual module is an *implementation* of this interface).

With web services, the interface (the set of "functions" you can call to access the data) takes the form of **HTTP Requests**—that is, a *request* for data sent following the *HyperText Transfer Protocol*. This is the same protocol (way of communicating) used by your browser to view a web page! An HTTP Request represents a message that your computer sends to a web server (another computer on the internet which "serves", or provides, information). That server, upon receiving the request, will determine what data to include in the **response** it sends *back* to the requesting computer. With a web browser the response data takes the form of HTML files that the browser can *render* as web pages; with data APIs the response data will be structured data that you can convert into structures such as lists or dictionaries.

In short, loading data from a Web API involves sending an **HTTP Request** to a server for a particular piece of data, and then receiving and parsing the **response** to that request.

12.2   RESTful Requests

There are two parts to a request sent to an API: the name of the **resource** (data) that you wish to access, and a **verb** indicating what you want to do with that resource. In many ways, the *verb* is the function you want to call on the API, and the *resource* is an argument to that function.

12.2.1   URIs

Which **resource** you want to access is specified with a **Uniform Resource Identifier (URI)**. A URI is a generalization of a URL (Uniform Resource Locator)—what you commonly think of as "web addresses". URIs act a lot like the *address* on a postal letter sent within a large organization such as a university: you indicate the business address as well as the department and the person, and will get a different response (and different data) from Alice in Accounting than from Sally in Sales.

- Note that the URI is the **identifier** (think: variable name) for the resource, while the **resource** is the actual *data* value that you want to access.

Like postal letter addresses, URIs have a very specific format used to direct the request to the right resource.

Not all parts of the format are required—for example, you don't need a `port`, `query`, or `fragment`. Important parts of the format include:

- `scheme` (`protocol`): the "language" that the computer will use to communicate the request to this resource. With web services this is normally `https` (secure HTTP)
- `domain`: the address of the web server to request information from

Figure 12.1: The format (schema) of a URI.

- `path`: which resource on that web server you wish to access. This may be the name of a file with an extension if you're trying to access a particular file, but with web services it often just looks like a folder path!
- `query`: extra **parameters** (arguments) about what resource to access.

The `domain` and `path` usually specify the resource. For example, `www.domain.com/users` might be an *identifier* for a resource which is a list of users. Note that web services can also have "subresources" by adding extra pieces to the path: `www.domain.com/users/joel` might refer to the specific "joel" user in that list.

With an API, the domain and path are often viewed as being broken up into two parts:

- The **Base URI** is the domain and part of the path that is included on *all* resources. It acts as the "root" for any particular resource. For example, the GitHub API has a base URI of `https://api.github.com/`.

- An **Endpoint**, or which resource on that domain you want to access. Each API will have *many* different endpoints.

  Note that most resources include multiple subresource endpoints. For example, you can access information about a specific user at the endpoint `/users/:username`—the colon `:` indicates that the subresource name is a *variable*—you can replace that word in the endpoint with whatever string you want. Thus if you were interested in the GitHub user `joelwross`, you would access the `/users/joelwross` endpoint.

  Subresources may have further subresources (which may or may not have variable names). The endpoint `/orgs/:org/repos` refers to the list of repositories belonging to an organization.

  Variable names in resources can also be written inside of curly braces `{}`; for example, `/orgs/{org}/repos`. Neither the colon nor the braces is programming language syntax, just common conventions used to communicate endpoints.

The **endpoint** is appended to the end of the **base URI**, so you could access a GitHub user by combining the **base URI** (`https://api.github.com`) and **endpoint** (`/users/joelwross`) into a single string: `https://api.github.com/users/joelwross`. That URL will return a data structure of information about the GitHub user,

which you can request from a Python program or simply view in your web-browser. Thus you can equivalently talk about accessing a particular **resource** and sending a request to a particular **endpoint**.

12.2.1.1   Query Parameters

Often in order to access only partial sets of data from a resource (e.g., to only get some users) you also include a set of **query parameters**. These are like extra arguments that are given to the request function. Query parameters are listed after a question mark **?** in the URI, and are formed as key-value pairs similar to how you named items in *lists*. The **key** (*parameter name*) is listed first, followed by an equal sign **=**, followed by the **value** (*parameter value*); note that you can't include any spaces in URIs! You can include multiple query parameters by putting an ampersand **&** between each key-value pair:

```
?firstParam=firstValue&secondParam=secondValue&thirdParam=thirdValue
```

Exactly what parameter names you need to include (and what are legal values to assign to that name) depends on the particular web service. Common examples include having parameters named `q` or `query` for searching, with a value being whatever term you want to search for: in `https://www.google.com/search?q=informatics`, the **resource** at the `/search` **endpoint** takes a query parameter `q` with the term you want to search for!

12.2.1.2   Access Tokens and API Keys

Many web services require you to register with them in order to send them requests. This allows them to limit access to the data, as well as to keep track of who is asking for what data (usually so that if someone starts "spamming" the service, they can be blocked).

To facilitate this tracking, many services provide **Access Tokens** (also called **API Keys**). These are unique strings of letters and numbers that identify a particular developer (like a secret password that only works for you). Web services will require you to include your *access token* as a query parameter in the request; the exact name of the parameter varies, but it often looks like `access_token` or `api_key`. When exploring a web service, keep an eye out for whether they require such tokens.

Access tokens act a lot like passwords; you will want to keep them secret and not share them with others. This means that you **should not include them in your committed files**, so that the passwords don't get pushed to GitHub and shared with the world. The best way to do this in Python is to create a separate script file in your repo (e.g., `apikeys.py`) which includes exactly one line: assigning the key to a variable:

```
## in `apikeys.py`
my_api_key = "123456789abcdefg"
```

You can then include this file*name* in a **.gitignore** file in your repo; that will keep it from even possibly being committed with your code!

In order to access this variable in your "main" script, you can **import** this file as a module. The module is the name of the file, and you can access specific variables from it to include them in the "global" scope. Note that importing a file as a module will execute each line of code in that module (that isn't in a "main" block):

```
## in `my_script.py`
from apikeys import my_api_key

print(my_api_key)  # key is now available!
```

Note that this assumes the `apikeys.py` file is inside the same folder as the script being run. See the documentation for details on how to handle other options.

Anyone else who runs the script will simply need to provide a `my_api_key` variable to access the API using their key, keeping everyone's accounts separate and private!

Watch out for APIs that mention using OAuth when explaining API keys. OAuth is a system for performing **authentification**—that is, letting you or a user log into a website from your application (like what a "Log in with Facebook" button does). OAuth systems require more than one access key, and these keys **_must_** be kept secret and usually require you to run a web server to utilize them correctly (which requires lots of extra setup, see `requests_oauthlib` library for details). So for this book, we encourage you to avoid anything that needs OAuth

12.2.2 HTTP Verbs

When you send a request to a particular resource, you need to indicate what you want to *do* with that resource. When you load web content, you are typically sending a request to retrieve information (logically, this is a `GET` request). However, there are other actions you can perform to modify the data structure on the server. This is done by specifying an **HTTP Verb** in the request. The HTTP protocol supports the following verbs:

- `GET` Return a representation of the current state of the resource
- `POST` Add a new subresource (e.g., insert a record)
- `PUT` Update the resource to have a new state
- `PATCH` Update a portion of the resource's state
- `DELETE` Remove the resource

- `OPTIONS` Return the set of methods that can be performed on the resource

By far the most common verb is `GET`, which is used to "get" (download) data from a web service. Depending on how you connect to your API (i.e., which programming language you are using), you'll specify the verb of interest to indicate what we want to do to a particular resource.

Overall, this structure of treating each datum on the web as a **resource** which we can interact with via **HTTP Requests** is referred to as the **REST Architecture** (REST stands for *REpresentational State Transfer*). This is a standard way of structuring computer applications that allows them to be interacted with in the same way as everyday websites. Thus a web service that enabled data access through named resources and responds to HTTP requests is known as a **RESTful** service, with a *RESTful API.*

12.3 Accessing Web APIs

To access a Web API, you just need to send an HTTP Request to a particular URI! You can easily do this with the browser: simple navigate to a particular address (base URI + endpoint), and that will cause the browser to send a GET request and display the resulting data in the browser. For example, you can send a request to search GitHub for repositories named `d3` by visiting:

`https://api.github.com/search/repositories?q=d3&sort=stars`

This query accesses the `/search/repositories/` endpoint, and also specifies 2 query parameters:

- `q`: The term(s) you are searching for, and
- `sort`: The attribute of each repository that you would like to use to sort the results

(Note that the data you'll get back is structured in JSON format. See below for details).

In Python you can most easily send GET requests using the `requests` module. This is a **third-party** library (not built into Python!), but is installed by default with Anaconda and so can just be imported:

```
import requests
```

This library provides a number of functions that reflect HTTP verbs. For example, the **get()** function will send an HTTP `GET` Request to the specified URI:

```
response = requests.get("https://api.github.com/search/repositories?q=d3&sort=stars"
```

While it is possible to include *query parameters* in the URI, `requests` also allows you to include them as a *dictionary*, making it easy to set and change

variables (instead of needing to do complex String concatenation):

```
resource_uri = "https://api.github.com/search/repositories"
query_params = {'q': 'd3', 'sort': 'stars'}
response = requests.get(resource_uri, params = query_params)
```

Pro tip: You can access the `response.url` property to see the URI where a request was sent to. This is useful for debugging: print out the URI that you constructed to paste it into your browser!

If you try printing out the `response` variable, you'll just see a simple output: `<Response [200]>`. The `200` is a an HTTP Status Code: an integer giving details about how the request went (`200` means "OK" the request was answered successfully. Other common codes are `404` "Not Found" if the resource isn't there, or `403` "Forbidden" if you don't have permission to access the resource, such as if your API key isn't specified correctly).

HTTP Status Codes are part of the **response header**. Each **response** has two parts: the **header**, and the **body**. You can think of the response as a envelope: the *header* contains meta-data like the address and postage date, while the *body* contains the actual contents of the letter (the data).

- You can view all of the headers in the `response.headers` property.

Since you're almost always interested in working with the *body*, you will need to extract that data from the response (e.g., open up the envelope and pull out the letter). You can get this content as the `text` property:

```
# extract content from response, as a text string
body = response.text
```

12.4 JSON Data

If you print out this text, you'll see what looks like an oddly-formatted dictionary. This is because most APIs will return data in **JavaScript Object Notation (JSON)** format. Like CSV, this is a format for writing down structured data—but while `.csv` files organize data into rows and columns, JSON allows you to organize elements into *sequences* and *key-value pairs*—similar to Python lists and dictionaries!

JSON format looks a lot like the literal format for Python *lists* and *dictionaries*. For example, consider the below JSON notation for Programmer Ada:

```
{
  "first_name": "Ada",
  "job": "Programmer",
  "salary": 78000,
  "in_union": true,
```

```
  "pets": ["rover", "fluffy", "mittens"],
  "favorites": {
    "music": "jazz",
    "food": "pizza",
    "numbers": [12, 42]
  }
}
```

JSON uses curly braces **{}** to define sets of key-value pairs (called *objects*, not "dictionaries"), and square brackets **[]** to define ordered sequences (called *arrays*, not "lists"). Like in Python, keys and values are separated with colons (**:**), and elements are separated by commas (**,**). As in Python, key-value pairs are often written on separate lines for readability, but this isn't required.

Note that unlike in Python dictionaries, keys *must* be strings (so in quotes), while values can either be strings, numbers, booleans (written in all lower-case as `true` and `false`), arrays (list), or other objects (dictionaries). Thus JSON can be seen as a way of writing normal Python dictionaries (or lists of dictionaries), just with a few restrictions on key and value type.

Pro-tip: JSON data can be quite messy when viewed in your web-browser. Installing a browser extension such as JSONView will format JSON responses in a more readable way, and even enable you to interactively explore the data structure.

JSON response data provided by the `requests` library can automatically be converted into the Python lists and dictionaries you are used to by using the **.json()** function:

```python
resource_uri = "https://api.github.com/search/repositories"
query_params = {'q': 'd3', 'sort': 'stars'}
response = requests.get(resource_uri, params = query_params)
data = response.json()
type(data)   # <class 'dict'>
             # may also be <class 'list'> for some resources
```

Note that it is also possible to use Python to convert from (and to!) a "JSON String" into standard data types like *list* or *dictionaries* using the **json** module:

```python
import json

# Convert to dictionary or list
data = json.loads(json_string)

# Convert to json
my_data = {'a':1, 'b':2}
my_json = json.dumps(my_data)
```

Note that these method names end in **s**: they are the "load **s**tring" and "dump **s**tring" methods (`load` and `dump` are for files!)

In practice, web APIs often return highly nested JSON objects (lots of dictionaries in dictionaries); you may need to go a "few levels deep" in order to access the data you really care about. Use the `keys()` dictionary method (or your web browser!) to inspect the returned data structure and find the values you are interested in. Note that it can be useful to "flatten" the data into a simpler structure for further access.

For example:

```python
resource_uri = "https://api.github.com/search/repositories"
query_params = {'q': 'd3', 'sort': 'stars'}
response = requests.get(resource_uri, params = query_params)
data = response.json()
print(data.keys())  # dict_keys(['total_count', 'incomplete_results', 'items'])
    # looking at the JSON data itself (e.g., in the browser), `items` is the key
    # that contains the value you want

result_items = data['items']  # "flatten" into new variable
print(len(result_items))  # 30, number of results returned
```

Resources

- URIs (Wikipedia)
- HTTP Protocol Tutorial
- Programmable Web (list of web APIs; may be out of date)
- RESTful Architecture (original specification; not for beginners)
- JSON View Extension
- Networked Programs (Downey); Using Web Services (Downey)
- `requests` documentation (quickstart)

Chapter 13

Git Branches

While `git` is great for uploading and downloading code, its true benefits are its ability to support *reversability* (e.g., undo) and *collaboration* (working with other people). In order to effectively utilize these capabilities, you need to understand git's **branching model**, which is central to how the program manages different versions of code.

This chapter will cover how to work with **branches** with git and GitHub, including using them to work on different features simultaneously and to undo previous changes.

13.1   Git Branches

So far, you've been using git to create a *linear sequence* of commits: they are all in a line, one after another).

Each commit has a message associated with it (that you can see with `git log --oneline`), as well as a unique SHA-1 hash (the random numbers and letters), which can be used to identify that commit as an "id number".

But you can also save commits in a *non-linear* sequence. Perhaps you want to try something new and crazy without breaking code that you've already written. Or you want to work on two different features simultaneously (having separate commits for each). Or you want multiple people to work on the same code without stepping on each other's toes.

To do this, you use a feature of git called **branching** (because you can have commits that "branch off" from a line of development):

In this example, you have a primary branch (called the `master` branch), and decide you want to try an experiment. You *split off* a new branch (called for

example `experiment`), which saves some funky changes to your code. But then you decide to make further changes to your main development line, adding more commits to `master` that ignore the changes stored in the `experiment` branch. You can develop `master` and `experiment` simultaneously, making changes to each version of the code. You can even branch off further versions (e.g., a `bugfix` to fix a problem) if you wish. And once you decide you're happy with the code added to both versions, you can **merge** them back together, so that the `master` branch now contains all the changes that were made on the `experiment` branch. If you decided that the `experiment` didn't work out, you can simply delete those set of changes without ever having messed with your "core" `master` branch.

You can view a list of current branches in the repo with the command

```
git branch
```

(The item with the asterisk (`*`) is the "current branch" you're on. The latest commit of the branch you're on is referred to as the **HEAD**.

You can use the same command to create a *new* branch:

```
git branch [branch_name]
```

This will create a new branch called `branch_name` (replacing `[branch_name]`, including the brackets, with whatever name you want). Note that if you run `git branch` again you'll see that this *hasn't actually changed what branch you're on*. In fact, all you've done is created a new *reference* (like a new variable!) that refers to the current commit as the given branch name.

- You can think of this like creating a new variable called `branch_name` and assigning the latest commit to that! Almost like you wrote `new_branch <- my_last_commit`.

- If you're familiar with LinkedLists, it's a similar idea to changing a pointer in those.

In order to switch to a different branch, use the command (without the brackets)

```
git checkout [branch_name]
```

Checking out a branch doesn't actually create a new commit! All it does is change the `HEAD` (the "commit I'm currently looking at") so that it now refers to the latest commit of the target branch. You can confirm that the branch has changed with `git branch`.

- You can think of this like assigning a new value (the latest commit of the target branch) to the `HEAD` variable. Almost like you wrote `HEAD <- branch_name_last_commit`.

- Note that you can create *and* checkout a branch in a single step using the `-b` option of `git checkout`:

```
git checkout -b [branch_name]
```

Once you've checked out a particular branch, any *new* commits from that point on will be "attached" to the "HEAD" of that branch, while the "HEAD" of other branches (e.g., `master`) will stay the same. If you use `git checkout` again, you can switch back to the other branch.

- **Important** checking out a branch will "reset" your code to whatever it looked like when you made that commit. Switch back and forth between branches and watch your code change!

Note that you can only check out code if the *current working directory* has no uncommitted changes. This means you'll need to `commit` any changes to the current branch before you `checkout` another. If you want to "save" your changes but don't want to commit to them, you can also use git's ability to temporarily stash changes.

Finally, you can delete a branch using `git branch -d [branch_name]`. Note that this will give you a warning if you might lose work; be sure and read the output message!

13.2 Merging

If you have changes (commits) spread across multiple branches, eventually you'll want to combine those changes back into a single branch. This is a process called **merging**: you "merge" the changes from one branch *into* another. You do this with the (surprise!) `merge` command:

```
git merge [other_branch]
```

This command will merge `other_branch` **into the current branch**. So if you want to end up with the "combined" version of your commits on a particular branch, you'll need to switch to (`checkout`) that branch before you run the merge.

IMPORTANT If something goes wrong, don't panic and try to close your command-line! Come back to this book and look up how to fix the problem you've encountered (e.g., how to exit *vim*). And if you're unsure why something isn't working with git, use **git status** to check the current status and for what steps to do next.

Note that the `rebase` command will perform a similar operation, but without creating a new "merge" commit—it simply takes the commits from one branch and attaches them to the end of the other. This effectively **changes history**, since it is no longer clear where the branching occurred. From an archival and academic view, you never want to "destroy history" and lose a record of changes

that were made. History is important: don't screw with it! Thus we recommend you *avoid* rebasing and stick with merging.

13.2.1   Merge Conflicts

Merging is a regular occurrence when working with branches. But consider the following situation:

1. You're on the `master` branch.
2. You create and `checkout` a new branch called `danger`
3. On the `danger` branch, you change line 12 of the code to be "I like kitties". You then commit this change (with message "Change line 12 of danger").
4. You `checkout` (switch to) the `master` branch again.
5. On the `master` branch, you change to line 12 of the code to be "I like puppies". You then commit this change (with message "Change line 12 of master").
6. You use `git merge danger` to merge the `danger` branch **into** the `master` branch.

In this situation, you are trying to *merge two different changes to the same line of code*, and thus should be shown an error on the command-line:

```
is-joelrossm13x:git_example joelross$ git merge danger
Auto-merging script.py
CONFLICT (content): Merge conflict in script.py
Automatic merge failed; fix conflicts and then commit the result.
```

Figure 13.1: A merge conflict reported on the command-line

This is called a **merge conflict**. A merge conflict occurs when two commits from different branches include different changes to the same code (they conflict). Git is just a simple computer program, and has no way of knowing which version to keep ("Are kitties better than puppies? How should I know?!").

Since git can't determine which version of the code to keep, it ***stops the merge in the middle*** and forces you to choose what code is correct **manually**.

In order to **resolve the merge conflict**, you will need to edit the file (code) so that you pick which version to keep. Git adds "code" to the file to indicate where you need to make a decision about which code is better:

In order to resolve the conflict:

1. Use `git status` to see which files have merge conflicts. Note that files may have more than one conflict!

2. Choose which version of the code to keep (or keep a combination, or replace it with something new entirely!) You do this by **editing the file** (i.e., open it in VS Code and change it). Pretend that your cat walked

Figure 13.2: Code including a merge conflict.

across your keyboard and added a bunch of extra junk; it is now your task to fix your work and restore it to a clean, working state. ***Be sure and test your changes to make sure things work!***

3. Be sure and remove the `<<<<<<<` and `=======` and `>>>>>>>`. These are not legal code in any language.

4. Once you're satisfied that the conflicts are all resolved and everything works as it should, follow the instructions in the error message and `add` and `commit` your changes (the code you "modified" to resolve the conflict):

```
git add .
git commit "Resolve merge conflict"
```

This will complete the merge! Use `git status` to check that everything is clean again.

Merge conflicts are expected. You didn't do something wrong if one occurs! Don't worry about getting merge conflicts or try to avoid them: just resolve the conflict, fix the "bug" that has appeared, and move on with your life.

13.3   Undoing Changes

One of the key benefits of version control systems is **reversibility**: the ability to "undo" a mistake (and we all make lots of mistakes when programming!) Git provides two basic ways that you can go back and fix a mistake you've made previously:

1. You can replace a file (or the entire project directory!) with a version saved as a previous commit.

2. You can have git "reverse" the changes that you made with a previous commit, effectively applying the *opposite* changes and thereby undoing it.

Note that both of these require you to have committed a working version of the code you want to go back to. Git only knows about changes that have been committed—if you don't commit, git can't help you! **Commit early, commit often**.

For both forms of undoing, first recall how each commit has a unique SHA-1 hash (those random numbers) that acted as its "name". You can see these with the `git log --oneline` command.

You can use the `checkout` command to switch not only to the commit named by a branch (e.g., `master` or `experiment`), but to *any* commit in order to "undo" work. You refer to the commit by its hash number in order to check it out:

```
git checkout [commit_number] [filename]
```

This will replace the current version ***of a single file*** with the version saved in `commit_number`. You can also use `--` as the commit-number to refer to the `HEAD` (the most recent commit in the branch):

```
git checkout -- [filename]
```

If you're trying to undo changes to lots of files, you can alternatively replace the entire project directory with a version from a previous commit by checking out that commit **as a new branch**:

```
git checkout -b [branch_name] [commit_number]
```

This command treats the commit as if it was the HEAD of a named branch… where the name of that branch is the commit number. You can then make further changes and merge it back into your development or `master` branch.

IMPORTANT NOTE: If you don't create a *new branch* (with `-b`) when checking out an old commit, you'll enter **detached HEAD state**. You can't commit from here, because there is no branch for that commit to be attached to! See this tutorial (scroll down) for details and diagrams. If you find yourself in a detached HEAD state, you can use `git checkout master` to get back to the last saved commit (though you will lose any changes you made in that detached state—so just avoid it in the first place!)

But what if you just had one bad commit, and don't want to throw out other good changes you made later? For this, you can use the `git revert` command:

```
git revert [commit_number] --no-edit
```

This will determine what changes that commit made to the files, and then apply the *opposite* changes to effectively "back out" the commit. Note that this **does**

not go back *to* the given commit number (that's what `checkout` is for!), but rather will *reverse the commit you specify.*

- This command does create a new commit (the `--no-edit` option tells git that you don't want to include a custom commit message). This is great from an archival point of view: you never "destroy history" and lose the record of what changes were made and then reverted. History is important: don't screw with it!

  Conversely, the `reset` command will destroy history. **Do not use it**, no matter what StackOverflow tells you to do.

13.4   GitHub and Branches

GitHub is an online service that stores copies of repositories in the cloud. When you `push` and `pull` to GitHub, what you're actually doing is **merging** your commits with the ones on GitHub!

However, remember that you don't edit any files on GitHub's servers, only on your own local machine. And since **resolving a merge conflict** involves editing the files, you have to be careful that conflicts only occur on the local machine, not on GitHub. This plays out in two ways:

1. You will **not** be able to **push** to GitHub if merging your commits ***into*** GitHub's repo would cause a merge conflict. Git will instead report an error, telling you that you need to `pull` changes first and make sure that your version is "up to date". Up to date in this case means that you have downloaded and merged all the commits on your local machine, so there is no chance of divergent changes causing a merge conflict when you merge by pushing.

2. Whenever you **pull** changes from GitHub, there may be a merge conflict! These are resolved ***in the exact same way*** as when merging local branches: that is, you need to *edit the files* to resolve the conflict, then `add` and `commit` the updated versions.

Thus in practice, when working with GitHub (and especially with multiple people), in order to upload your changes you'll need to do the following:

1. `pull` (download) any changes you don't have
2. *Resolve* any merge conflicts that occurred
3. `push` (upload) your merged set of changes

Additionally, because GitHub repositories are repos just like the ones on your local machine, they can have branches as well! You have access to any *remote* branches when you `clone` a repo; you can see a list of them with `git branch -a` (using the "**a**ll" option).

If you create a new branch on your local machine, it is possible to push *that branch* to GitHub, creating a mirroring branch on the remote repo. You do this by specifying the branch in the `git push` command:

```
git push origin branch_name
```

where `branch_name` is the name of the branch you are currently on (and thus want to push to GitHub).

Note that you often want to associate your local branch with the remote one (make the local branch **track** the remote), so that when you use `git status` you will be able to see whether they are different or not. You can establish this relationship by including the `-u` option in your push:

```
git push -u origin branch_name
```

Tracking will be remembered once set up, so you only need to use the `-u` option *once.*

13.4.1   GitHub Pages

GitHub's use of branches provides a number of additional features, one of which is the ability to **host** web pages (`.html` files, which can be generated from Markdown or Jupyter notebooks) on a publicly accessible web server that can "serve" the page to anyone who requests it. This feature is known as GitHub Pages.

With GitHub pages, GitHub will automatically serve your files to visitors as long as the files are in a branch with a magic name: **gh-pages**. Thus in order to **publish** your webpage and make it available online, all you need to do is create that branch, merge your content into it, and then push that branch to GitHub.

You almost always want to create the new `gh-pages` branch off of your `master` branch. This is because you usually want to publish the "finished" version, which is traditionally represented by the `master` branch. This means you'll need to switch over to `master`, and then create a new branch from there:

```
git checkout master
git checkout -b gh-pages
```

Checking out the new branch will create it *with all of the commits of its source* meaning `gh-pages` will start with the exact same content as `master`—if your page is done, then it is ready to go!

You can then upload this new local branch to the `gh-pages` branch on the `origin` remote:

```
git push -u origin gh-pages
```

After the push completes, you will be able to see your web page using the following URL:

```
https://GITHUB-USERNAME.github.io/REPO-NAME
```

(Replace `GITHUB-USERNAME` with the user name **of the account hosting the repo**, and `REPO-NAME` with your repository name).

- This means that if you're making your homework available, the `GITHUB-USERNAME` will be the name of the course organization.

Some important notes:

1. The `gh-pages` branch must be named *exactly* that. If you misspell the name, or use an underscore instead of a dash, it won't work.

2. Only the files and commits in the `gh-pages` branch are visible on the web. All commits in other branches (`experiment`, `master`, etc.) are not visible on the web (other than as source code in the repo). This allows you to work on your site with others before publishing those changes to the web.

3. Any content in the `gh-pages` branch will be publicly accessible, even if your repo is private. You can remove specific files from the `gh-pages` branch that you don't want to be visible on the web, while still keeping them in the `master` branch: use the `git rm` to remove the file and then add, commit, and push the deletion.

   - Be careful not push any passwords or anything to GitHub!

4. The web page will only be initially built when a **repo administrator** pushes a change to the `gh-pages` branch; if someone just has "write access" to the repo (e.g., they are a contributor, but not an "owner"), then the page won't be created. But once an administrator (such as the person who created the repo) pushes that branch and causes the initial page to be created, then any further updates will appear as well.

After you've created your initial `gh-pages` branch, any changes you want to appear online will need to be saved as new commits to that branch and then pushed back up to GitHub. **HOWEVER**, it is best practice to *not* make any changes directly to the `gh-pages` branch! Instead, you should switch back to the `master` branch, make your changes there, commit them, then `merge` them back into `gh-pages` before pushing to GitHub:

```
# switch back to master
git checkout master

### UPDATE YOUR CODE (outside of the terminal)

# commit the changes
```

```
git add .
git commit -m "YOUR CHANGE MESSAGE"

# switch back to gh-pages and merge changes from master
git checkout gh-pages
git merge master

# upload to github
git push --all
```

(the `--all` option on `git push` will push all branches that are **tracking** remote branches).

This procedure will keep your code synchronized between the branches, while avoiding a large number of merge conflicts.

Resources

- Git and GitHub in Plain English
- Atlassian Git Branches Tutorial
- Git Branching (Official Documentation)
- Learn Git Branching (interactive tutorial)
- Visualizing Git Concepts (interactive visualization)
- Resolving a merge conflict (GitHub)

Chapter 14

Git Collaboration

Being able to merge between branches allows you to work **collaboratively**, with multiple people making changes to the same repo and sharing those changes through GitHub. There are a variety of approaches (or **workflows**) that can be used to facilitate collaboration and make sure that people are effectively able to share code. This section describes a variety of different workflows; however we suggest the branch-based workflow called the **Feature Branch Workflow** for this course.

14.1 Centralized Workflow

In order to understand the Feature Branch Workflow, it's important to first understand how to collaborate on a centralized repository. The Feature Branch Workflow uses a **centralized repository** stored on GitHub—that is, every single member of the team will push and pull to a single GitHub repo. However, since each repository needs to be created under a particular account, this means that a *single member* of the team will need to create the repo (such as by accepting a GitHub Classroom assignment, or by clicking the *"New"* button on their "Repositories" tab on the GitHub web portal).

In order to make sure everyone is able to push to the repository, whoever creates the repo will need to **add the other team members as collaborators**. You can do this under the **Settings** tab:

Once you've added everyone to the GitHub repository, **each team member** will need to **clone** the repository to their local machines to work on the code individually. Collaborators can then push any changes they make to the central repository, and pull and changes made by others. Because multiple members will be contributing to the *same repositiory*, it's important to ensure that you are working on the most up-to-date version of the code. This means that you
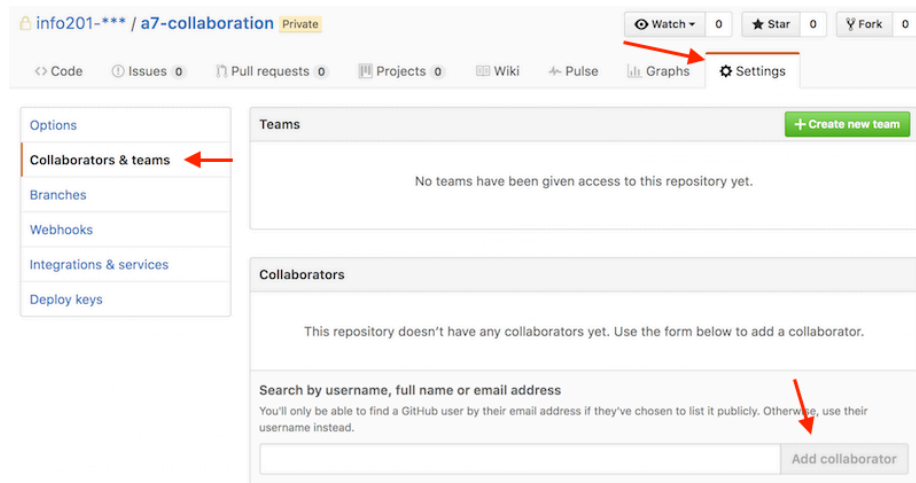
Figure 14.1: Adding a collaborator to a Github repo (via the web portal).

will regularly have to **pull in changes** from GitHub that your team members may have committed. As a result, we suggest that you have a workflow that looks like this:

```
# Begin your work session by pulling in changes from GitHub
git pull origin master

# If necessary, resolve any merge conflicts and commit them
git add .
git commit --no-edit  # use the default merge commit message

# Do your own work here!

# Then commit and push that work
git add .
git commit -m "Make progress on feature X"
git push origin master
```

Note that if someone pushes a commit to GitHub *before you push your changes*, you'll need to integrate those changes into your code (and test them!) before pushing up to GitHub. While working on a single `master` branch in this fashion is possible, you'll encounter fewer conflicts if you use a dedicated **feature branch** for each developer or feature you're working on.

14.2 Feature Branch Workflow

The Feature Branch Workflow is a natural extension of the Centralized Workflow that enhances the model by defining specific *branches* for different pieces of development (still with one centralized repository). The core idea behind the Feature Branch Workflow is that all development should take place on a dedicated **feature branch**, rather than on the `master` branch. This allows for different people to work on different branches without disturbing the main codebase. For example, you might have one branch `visualization` that focuses on adding a complex visualization, or another `experimental-analysis` that tries a bold new approach to processing the data. Each branch is based on a *feature* (capability or part) of the project, not a particular person: a single developer could be working on multiple feature branches.

The idea is that the `master` branch *always* contains "production-level" code: valid, completely working code that you could deploy or publish (read: give to your boss or teacher) at a whim. All feature branches branch off of `master`, and are allowed to contain temporary or even broken code (since they are still in development). This way there is always a "working" (if incomplete) copy of the code (`master`), and development can be kept isolated and considered independent of the whole. This is similar to the example with the `experiment` branch in Chapter 14.

The workflow thus works like this:

1. Ada decides to add a new feature or part to the code. She creates a new feature branch off of `master`:

   ```
   git checkout master
   git checkout -b adas-feature
   ```

2. Ada does some work on this feature

   ```
   # work is done outside of terminal

   git add .
   git commit -m "Add progress on feature"
   ```

3. Ada takes a break, pushing her changes to GitHub

   ```
   git push -u origin adas-feature
   ```

4. After talking to Ada, Bebe decides to help finish up the feature. She checks out the branch and makes some changes, then pushes them back to GitHub

   ```
   # fetch will "download" commits from GitHub, without merging them
   git fetch origin
   git checkout adas-feature
   ```

```
# Work is on adas-feature done outside of terminal

git add .
git commit -m "Add more progress on feature"
git push origin adas-feature
```

5. Ada downloads Bebe's changes

```
git pull origin adas-feature
```

6. Ada decides the feature is finished, and *merges* it back into `master`. But first, she makes sure she has the latest version of the `master` code to integrate her changes with

```
git checkout master  # switch to master
git pull origin master  # download any changes

git merge adas-feature  # merge the feature into the master branch
# fix any merge conflicts!!

git push origin master  # upload the updated code to master
```

7. And now that the feature has been successfully added to the project, Ada can delete the feature branch (using `git branch -d branch_name`). See also here.

This kind of workflow is very common and effective for supporting collaboration. Note that as projects get large, you may need to start being more organized about how and when you create feature branches. For example, the **Git Flow** model organizes feature branches around product releases, and is often a starting point for large collaborative projects.

14.3   Forking Workflow

The Forking Workflow takes a **fundamentally different approach** to collaboration than the Centralized and Feature Branch workflows. Rather than having a single remote, each developer will have **their own repository** on GitHub that is *forked* from the original repository. As discussed in the introductory GitHub Chapter, a developer can create their own remote repository from an existing project by *forking* it on GitHub. This allows the individual to make changes (and contribute to) the project. However, we have not yet discussed how those changes can be integrated into the original code base. GitHub offers a feature called **pull requests** by which you can merge two remote branches (that is: `merge` two branches that are on GitHub). A **pull request** is a request for the changes from one branch to be pulled (merged) into another.

14.3.1 Pull Requests

Pull requests are primarily used to let teams of developers *collaborate*—one developer can send a request "hey, can you integrate my changes?" to another. The second developer can perform a **code review**: reviewing the proposed changes and making comments or asking for corrections to anything they find problematic. Once the changes are improved, the pull request can be **accepted** and the changes merged into the target branch. This process is how programmers collaborate on *open-source software* (such as Python libraries like `requests`): a developer can *fork* an existing professional project, make changes to that fork, and then send a pull request back to the original developer asking them to merge in changes ("will you include my changes in your branch/version of the code?").

Pull requests should only be used when doing collaboration using remote branches! Local branches should be `merged` locally using the command line, not GitHub's pull request feature.

In order to issue a pull request, both branches you wish to merge will need to be `pushed` to GitHub (whether they are in the same repo or in forks). To issue the pull request, navigate to your repository on GitHub's web portal and choose the **New Pull Request** button (it is next to the drop-down that lets you view different branches).

In the next page, you will need to specify which branches you wish to merge. The **base** branch is the one you want to merge *into* (often `master`), and the **head** branch (labeled "compare") is the branch with the new changes you want to merge (often a feature branch).

Add a title and description for your pull request. These should follow the format for git commit messages. Finally, click the **Create pull request** button to finish creating the pull request.

Important! The pull request is a request to merge two branches, not to merge a specific set of commits. This means that you can *push more commits* to the head/merge-from branch, and they will automatically be included in the pull request—the request is always "up-to-date" with whatever commits are on the (remote) branch.

You can view all pull requests (including those that have been accepted) through the **Pull Requests** tab at the top of the repo's web portal. This is where you can go to see comments that have been left by the reviewer.

If someone sends you a pull request (e.g., another developer on your team), you can accept that pull request through GitHub's web portal. If the branches can be merged without a conflict, you can do this simply by hitting the **Merge pull request** button. However, if GitHub detects that a conflict may occur, you will need to pull down the branches and merge them locally.

It is best practice to *never* accept your own pull requests! If you don't need any

collaboration, just merge the branches locally.

Note that when you merge a pull request via the GitHub web site, the merge is done entirely on the server. Your local repo will not yet have those changes, and so you will need to use `git pull` to download the updates to an appropriate branch.

Resources

- Atlassian Git Workflows Tutorial

Chapter 15

Object-Oriented Programming

So far, this book has discussed using a **procedural programming** style to write computer programs. Procedural programming focuses on defining functions (*procedures*) as sequences of instructions to execute. While you've used *control structures* (e.g., conditionals and loops) to redirect the program flow, and *data structures* (e.g., lists and dictionaries) to organize data, you have designed programs primarily based on which lines of code execute in what order.

Object-oriented programming is a programming paradigm (style) in which you don't design programs around the sequence of statements executed, but around the specification of **objects** which are combinations of *data* (variables) and *behaviors* (functions). Objects are used to represent elements in the program's real-world problem domain, and provide a useful *abstraction* when designing programs: instead of needing to think about code statements executing in order, you can think about telling "things" to perform "actions".

This chapter will discuss the fundamentals of object-oriented programming and how to specify **classes** (custom data types) that are used to define and create **objects**.

This chapter is under development. More details coming soon...

Resources

- Classes and Objects - The Basics (Downey)
- Classes and Objects - Digging a little deeper (Downey)
- Even more OOP (Downey)

Chapter 16

The **pandas** Library

This chapter introduces the *Python Data Analysis* library **pandas**—a set of modules, functions, and classes used to for easily and efficiently performing data analysis—`panda`'s speciality is its highly optimized performance when working with large data sets. `pandas` is the most common library used with Python for Data Science (and mirrors the `R` language in many ways, allowing programmers to easily move between the two). This chapter will discuss the two main data structures used by `pandas` (*Series* and *DataFrames*) and how to use them to organize and work with data.

16.1 Setting up **pandas**

`pandas` is a **third-party** library (not built into Python!), but is included by default with Anaconda and so can be imported directly. Additionally, Pandas is built on top of the `numpy` scientific computing library which supports highly optimized mathematical operations. Thus many `pandas` operations involve working with `numpy` data structures, and the `pandas` library requires `numpy` (also included in Anaconda) to also be imported:

```
# import libraries
import pandas as pd   # standard shortcut names
import numpy as np
```

Normal practice is to `import` the module and reference types and methods using dot notation, rather than importing them into the global namespace. Also note that this module will focus primarily on `pandas`, leaving `numpy`-specific data structures and functions for the reader to explore.

151

16.2   Series

The first basic `pandas` data structure is a **Series**. A Series represents a *one-dimensional ordered collection of values*, making them somewhat similar to a regular Python *list*. However, elements can also be given *labels* (called the **index**), which can be non-numeric values, similar to a *key* in a Python *dictionary*. This makes a Series somewhat like an ordered dictionary—one that supports additional methods and efficient data-processing behaviors.

Series can be created using the `Series()` function (a *constructor* for instances of the class):

```python
# create a Series from a list
number_series = pd.Series([1, 2, 2, 3, 5, 8])
print(number_series)
```

This code would print out:

```
0    1
1    2
2    2
3    3
4    5
5    8
dtype: int64
```

Printing a Series will display it like a *table*: the first value in each row is the **index** (label) of that element, and the second is the value of the element in the Series. Printing will also display the *type* of the elements in the Series. All elements in the Series will be treated as "same" type—if you create a Series from mixed elements (e.g., numbers and strings), the type will be the a generic `object`. In practice, you almost always create Series from a single type.

If you create a Series from a list, each element will be given an *index* (label) that is that values's index in the list. You can also create a Series from a *dictionary*, in which case the keys will be used as the index labels:

```python
# create a Series from a dictionary
age_series = pd.Series({'sarah': 42, 'amit': 35, 'zhang': 13})
print(age_series)
```

```
amit     35
sarah    42
zhang    13
dtype: int64
```

Notice that the Series is automatically **sorted** by the keys of the dictionary! This means that the order of the elements in the Series will always be the same for a given dictionary (which cannot be said for the dictionary items themselves).

16.2.1 Series Operations and Methods

The main benefit of Series (as opposed to normal lists or dictionaries) is that they provide a number of operations and methods that make it easy to consider and modify the entire Series, rather than needing to work with each element individually. These functions include built-in *mapping*, *reducing*, and *filtering* style operations.

In particular, basic operators (whether math operators such as + and -, or relational operators such as > or ==) function as **vectorized operations**, meaning that they are applied to the entire Series **member-wise**: the operation is applied to the first element in the Series, then the second, then the third, and so forth:

```python
sample = pd.Series(range(1,6))  # Series of numbers from 1 to 5 (6 is excluded)
result = sample + 4  # add 4 to each element (produces new Series)
print(result)
    # 0    5
    # 1    6
    # 2    7
    # 3    8
    # 4    9
    # dtype: int64

is_greater_than_3 = sample > 3  # compare each element
print(is_greater_than_3)
    # 0    False
    # 1    False
    # 2    False
    # 3     True  # note index and value are not the same
    # 4     True
    # dtype: bool
```

Having a Series operation apply to a *scalar* (a single value) is referred to as **broadcasting**. The idea is that the smaller "set" of elements (e.g., a single value) is *broadcast* so that it has a comparable size, thereby allowing different "sized" data structures to interact. Technically, operating on a Series with a *scalar* is actually a specific case of operating on it with another Series!

If the second operand is *another Series*, then mathematical and relational operations are still applied **member-wise**, with the elements of each operand being "matched" by their index label. This means that for most Series whose indices are list indices, operators will be applied "in order".

```python
s1 = pd.Series([2, 2, 2, 2, 2])
s2 = pd.Series([1, 2, 3, 4, 5])

# Examples of operations (list only includes values)
```

```
list(s1 + s2)   # [3, 4, 5, 6, 7]
list(s1 / s2)   # [2.0, 1.0, 0.66666666666666663, 0.5, 0.40000000000000002]
list(s1 < s2)   # [False, False, True, True, True]

# Add a Series to itself (why not?)
list(s2 + s2)   # [2, 4, 6, 8, 10]

# Perform more advanced arithmetic!
s3 = (s1 + s2) / (s1 + s1)
list(s3)  # [0.75, 1.0, 1.25, 1.5, 1.75]
```

And note that these operations will be *fast*, even for very large Series, allowing for effective data manipulations.

`pandas` Series also include a number of *methods* for inspecting and manipulating their data. Some useful examples are detailed below (not a comprehensive listing):

Function	Description
`index`	an *attribute*; the sequence of index labels (convert to a *list* to use)
`head(n)`	returns a Series containing only the first `n` elements
`tail(n)`	returns a Series containing only the last `n` elements
`any()`	returns whether ANY of the elements are `True` (or "truthy")
`all()`	returns whether ALL of the elements are `True` (or "truthy")
`mean()`	returns the statistical mean of the elements in the Series
`std()`	returns the standard deviation of the elements in the Series
`describe()`	returns a Series of descriptive statistics
`idxmax()`	returns the index label of the element with the max value

Series support many more methods as well: see the full documentation for a complete list.

One particularly useful method to mention is the `apply()` method. This method is used to *apply* a particular **callback function** to each element in the series. This is a *mapping* operation, similar to what you've done with the `map()` function:

```
def square(n):  # a function that squares a number
    return n**2


number_series = pd.Series([1,2,3,4,5])  # an initial series

square_series = number_series.apply(square)
list(square_series)  # [1, 4, 9, 16, 25]
```

```python
# Can also apply built-in functions
import math
sqrt_series = number_series.apply(math.sqrt)
list(sqrt_series)  # [1.0, 1.4142135623730951, 1.7320508075688772, 2.0, 2.2360679774997898]

# Pass additional arguments as keyword args (or `args` for a single argument)
cubed_series = number_series.apply(math.pow, args=(3,)) # call math.pow(n, 3) on each
list(cubed_series)  # [1.0, 8.0, 27.0, 64.0, 125.0]
```

16.2.2 Accessing Series

Just like lists and dictionaries, elements in a Series can be accessed using **bracket notation**, putting the index label inside the brackets:

```python
number_series = pd.Series([1, 2, 2, 3, 5, 8])
age_series = pd.Series({'sarah': 42, 'amit': 35, 'zhang': 13})

# Get the 1th element from the number_series
number_series[1]  # 2

# Get the 'sarah' element from age_series
age_series['amit']  # 35

# Get the 0th element from age_series
# (Series are ordered, so can always be accessed positionally)
age_series[0]  # 42
```

Note that the returned values are not technically basic `int` or `float` or `string` types, but are rather specific `numpy` objects that work almost identically to their normal type (but with some additional optimization).

You can also use list-style *slices* using the colon operator (e.g., elements `1:3`). Additionally, it is possible to specify ***a sequence of indicies*** (i.e., a *list* or *range* or even a *Series* of indices) to access using bracket notation. This will produce a new Series object that contains only the elements that have those labels:

```python
age_series = pd.Series({'sarah': 42, 'amit': 35, 'zhang': 13})

index_list = ['sarah', 'zhang']
print(age_series[index_list])
    # sarah    42
    # zhang    13
    # dtype: int64
```

```python
# using an anonymous variable for the index list (notice the brackets!)
print(age_series[['sarah', 'zhang']])
    # sarah    42
    # zhang    13
    # dtype: int64
```

This also means that you can use something like a *list comprehension* (or even a Series operation!) to determine which elements to select from a Series!

```python
letter_series = pd.Series(['a','b','c','d','e','f'])
even_numbers = [num for num in range(0,6) if num%2 == 0]  # list of even numbers

# Get letters with even numbered indices
letter_series[even_numbers]  # []
    # 0    a
    # 2    c
    # 4    e
    # dtype: object

# In one line (check the brackets!)
letter_series[[num for num in range(0,6) if num%2 == 0]]
```

Finally, using a **sequence of booleans** with bracket notation will produce a new Series containing the elements whose position *corresponds* with `True` values. This is called **boolean indexing**.

```python
shoe_sizes = pd.Series([7, 6.5, 4, 11, 8])  # a series of shoe sizes
index_filter = [True, False, False, True, True]  # list of which elements to extract

# Extract every element in an index that is True
shoe_sizes[index_filter]  # has values 7.0, 11.0, 8.0
```

In this example, since `index_filter` is `True` at index 0, 3, and 4, then `shoe_sizes[index_filter]` returns a Series with the elements from index numbers 0, 3, and 4.

This technique is incredibly powerful because it allows you to easily perform **filtering** operations on a Series:

```python
shoe_sizes = pd.Series([7, 6.5, 4, 11, 8])  # a series of shoe sizes
big_sizes = shoe_sizes > 6.5  # has values True, False, False, True, True

big_shoes = shoe_sizes[big_sizes]  # has values 7, 11, 8

# In one line
big_shoes = shoe_sizes[shoe_sizes > 6.5]
```

You can think of the last statement as saying *shoe sizes **where** shoe size is*

greater than 6.5.

Note that you can include *logical operators* ("and" and "or") by using the operators & for "and" and | for "or". Be sure to wrap each relational expression in () to enforce order of operations.

While it is perfectly possible to do similar filtering with a list comprehension, the boolean indexing expression can be very simple to read and runs quickly. (This is also the normal style of doing filtering in the R programming language).

16.3 DataFrames

The most common data structure used in `pandas` (more common than Series) is a **DataFrame**. A DataFrame represents a **table**, where data is organized into rows and columns. You can think of a DataFrame as being like a Excel spreadsheet or a SQL table. This book has previously represented tabular data using a *list of dictionaries*. However, this required you to be careful to make sure that all of the dictionaries shared keys, and did not offer easy ways to interact with the table in terms of its rows or columns. DataFrames give you that functionality!

A DataFrame can also be understood as a *dictionary of Series*, where each Series represents a **column** of the table. The keys of this dictionary are the *index labels* of the columns, while the the index labels of the Series serve as the labels for the row.

This is distinct from spreadsheets or SQL tables, which are often seen as a collection of *observations* (rows). Programmatically, DataFrames should primarily be considered as a collection of *features* (columns), which happen to be sequenced to correspond to observations.

A DataFrame can be created using the `DataFrame()` function (a *constructor* for instances of the class). This function usually takes as an argument a *dictionary* whose values are Series (or values that can be converted into a Series, such as a list or a dictionary):

```
name_series = pd.Series(['Ada', 'Bob', 'Chris', 'Diya', 'Emma'])
heights = [64, 74, 69, 69, 71]
weights = [135, 156, 139, 144, 152]

df = pd.DataFrame({'name': name_series, 'height': heights, 'weight': weights})
print(df)
    #      height    name  weight
    # 0        64     Ada     135
    # 1        74     Bob     156
    # 2        69   Chris     139
```

```
# 3       69    Diya      144
# 4       71    Emma      152
```

Although DataFrames variables are often named `df` in `pandas` examples, this is ***not*** a good variable name. You should use much more descriptive names for your DataFrames (e.g., `person_size_table`) when used in actual programs.

Note that you can specify the order of columns in the table using the `columns` keyword argument, and the order of the rows using the `index` keyword argument.

It is also possible to create a DataFrame directly from a spreadsheet—such as from **.csv** file (containing **c**omma **s**separated **v**alues) by using the `pandas.read_csv()` function:

```
my_dataframe = pd.read_csv('path/to/my/file.csv')
```

See the IO Tools documentation for details and other file-reading functions.

16.3.1   DataFrame Operations and Methods

Much like Series, DataFrames support a **vectorized** form of mathematical and relational operators: when the other operand is a *scalar*, then the operation is applied member-wise to each value in the DataFrame:

```
# data frame of test scores
test_scores = pd.DataFrame({
    'math':[91, 82, 93, 100, 78, 91],
    'spanish':[88, 79, 77, 99, 88, 93]
})

curved_scores = test_scores * 1.02  # curve scores up by 2%
print(curved_scores)
    #        math   spanish
    # 0    92.82     89.76
    # 1    83.64     80.58
    # 2    94.86     78.54
    # 3   102.00    100.98
    # 4    79.56     89.76
    # 5    92.82     94.86

print(curved_scores > 90)
    #      math  spanish
    # 0    True    False
    # 1   False    False
    # 2    True    False
    # 3    True     True
```

```
# 4   False    False
# 5    True     True
```

It is possible to have both operands be DataFrames. In this case, the operation is applied **member-wise**, where values are matched if they have the same row *and* column label. Note that any value that doesn't have a pair will instead produce the value `NaN` (Not a Number). This is not a normal way of working with DataFrames—it is much more common to access individual rows and columns and work with those (e.g., make a new column that is the sum of two others); see below for details.

Also like Series, DataFrames objects support a large number of methods, including:

Function	Description
`index`	an *attribute*; the sequence of **row** index labels (convert to a *list* to use)
`columns`	an *attribute*; the sequence of **column** index labels (convert to a *list* to use)
`head(n)`	returns a DataFrame containing only the first n *rows*
`tail(n)`	returns a DataFrame containing only the last n *rows*
`assign(...)`	returns a new DataFrame with an additional column; call as `df.assign(new_label=new_column)`
`drop(label, row_or_col)`	returns a new DataFrame with the given row or column removed
`mean()`	returns a Series of the statistical means of the values of each **column**
`all()`	returns a Series of whether ALL the elemnts in each **column** are `True` (or "truthy")
`describe()`	returns a DataFrame whose columns are Series of descriptive statistics for each **column** in the original DataFrame

You may notice that many of these methods (e.g., `head()`, `mean()`, `describe()`, `any()`) also exist for Series. In fact, most every method that Series support are supported by DataFrames as well. These methods are all applied **per column** (not per row)—that is, calling `mean()` on a DataFrame will calculate the *mean* of **each column** in that DataFrame:

```python
df = pd.DataFrame({
    'name': ['Ada', 'Bob', 'Chris', 'Diya', 'Emma'],
    'height': [64, 74, 69, 69, 71],
    'weights': [135, 156, 139, 144, 152]
    })
df.mean()
```

```
# height      69.4
# weights    145.2
# dtype: float64
```

If the Series method would return a *scalar* (a single value, as with `mean()` or `any()`), then the DataFrame method returns a Series whose labels are the column labels, as above. If the Series method instead would return a Series (multiple values, as with `head()` or `describe()`), then the DataFrame method returns a new DataFrame whose columns are each of the resulting Series:

```
df = pd.DataFrame({
    'name': ['Ada', 'Bob', 'Chris', 'Diya', 'Emma'],
    'height': [64, 74, 69, 69, 71],
    'weights': [135, 156, 139, 144, 152]
    })
df.describe()
    #            height      weights
    # count    5.000000     5.000000
    # mean    69.400000   145.200000
    # std      3.646917     8.757854
    # min     64.000000   135.000000
    # 25%     69.000000   139.000000
    # 50%     69.000000   144.000000
    # 75%     71.000000   152.000000
    # max     74.000000   156.000000
```

Notice that the `height` column is the result of calling `describe()` on the DataFrame's `height` column Series!

As a general rule: if you're expecting one value per column, you'll get a Series of those values; if you're expecting multiple values per column, you'll get a DataFrame of those values.

This also means that you can sometimes "double-call" methods to reduce them further. For example, `df.any()` returns a Series of whether each column contains a `True` value; `df.all().all()` would check if *that* Series contains all `True` values (thus checking *all* columns have all `True` value, i.e., the entire table is all `True` values).

16.3.2   Accessing DataFrames

DataFrames make it possible to quickly access individual or a subset of values, though these methods use a variety of syntax structures. For this explanation, refer to the following sample DataFrame initially described above:

```
# all examples in this section
df = pd.DataFrame({
```

```
    'name': ['Ada', 'Bob', 'Chris', 'Diya', 'Emma'],
    'height': [64, 74, 69, 69, 71],
    'weights': [135, 156, 139, 144, 152]
    })

print(df)
    #      height    name    weight
    # 0        64     Ada       135
    # 1        74     Bob       156
    # 2        69   Chris       139
    # 3        69    Diya       144
    # 4        71    Emma       152
```

Since DataFrames are most commonly viewed as a *dictionary of columns*, it is possible to access them as such using **bracket notation** (using the index label of the column):

```
print( df['height'] )   # get height column
    # 0      64
    # 1      74
    # 2      69
    # 3      69
    # 4      71
    # Name: height, dtype: int64
```

However, it is often more common to refer to individual columns using **dot notation**, treating each column as an *attribute* or *property* of the DataFrame object:

```
# same results as above
print( df.height )   # get height column
```

It is also possible to select *multiple* columns by using a *list* or sequence inside the **bracket notation** (similar to selecting multiple values from a Series). This will produce a new DataFrame (a "sub-table")

```
# count the brackets carefully!
print( df[['name', 'height']] )   # get name and height columns

# can also select multiple columns with a list of their positions
print( df[[1,2]] )   # get 1st (name) and 2nd (weight) columns
```

Watch out though! Specifying a **slice** (using a colon **:**) will actually select by *row* position, not column position! I do not know wherefore this inconsistency, other than "convenience".

```
print( df[0:2] ) # get ROWS 0 through 2 (not inclusive)
    #    height name   weight
```

```
# 0        64   Ada       135
# 1        74   Bob       156
```

Because DataFrames support multiple indexes, it is possible to use **boolean indexing** (as with Series), allowing you to *filter* for rows based the values in their columns:

```
print( df[df.height > 70] )
    #     height   name   weight
    # 1       74    Bob      156
    # 4       71   Emma      152
```

Note that `df.height` is a Series (a column), so `df.height > 70` produces a Series of boolean values (`True` and `False`). This Series is used to determine *which* rows to return from the DataFrame—each row that corresponds with a `True` index.

Finally, DataFrames also provide two *attributes* (properties) used to "quick access" values: **loc**, which provides an "index" (lookup table) based on index labels, and **iloc**, which provides an "index" (lookup table) based on row and column positions. Each of these "indexes" can be thought of as a *dictionary* whose values are the individual elements in the DataFrame, and whose keys can therefore be used to access those values using **bracket notation**. The dictionaries support multiple types of keys (using label-based `loc` as an example):

Key Type	Description	Example
`df.loc[row_label]`	An individual value	`df.loc['Ada']` (the row labeled `Ada`)
`df.loc[row_label_list]`	A list of row labels	`df.loc[['Ada','Bob']]` (the rows labeled `Ada` and `Bob`)
`df.loc[row_label_slice]`	A *slice* of row labels	`df.loc['Bob':'Diya']` (the rows from `Bob` to `Diya`. Note that this is an *inclusive* slice!)
`df.loc[row_label, col_label]`	A *tuple* of (row, column)	`df.loc['Ada', 'height']` (the value at row `Ada`, column `height`)
`df.loc[row_label_seq, col_label_seq]`	A *tuple* of label lists or slices	`df.loc['Bob':'Diya', ['height','weight']]` (the rows from `Bob` to `Diya` with the columns `height` and `weight`)

- Note that the example `df` table doesn't have row labels beyond `0` to `4`

- Using a *tuple* makes it easy to access a particular value in the table, or a range of values (*selecting* rows and columns).

- Note that we can also use the boundless slice `:` to refer to "all elements". So for example:

```
df.loc[:, 'height']  # get all rows, 'height' column
```

This is a basic summary of how to create and access DataFrames; for more detailed usage, additional methods, and specific "recipes", see the official `pandas` documentation.

Resources

- 10 minutes to pandas (pandas docs) a basic set of examples
- Tutorials (pandas docs) a list and guide to various tutorials (of mixed quality)
- Intro to Data Structure (pandas docs)
- Essential Basic Functionality (pandas docs) not really basic, but a complete set of examples
- Pandas. Data Processing (Data Analysis in Python)
- pandas Foundations (DataCamp)

Chapter 17

JavaScript

JavaScript for Snake People

JavaScript is a popular, high-level, general-purpose programming language primarily used for implementing interactive web applications and other information systems. JavaScript shares many structural (and even some syntactical) similarities with Python; the places where it differs often draws from other major major languages like Java and C (and thus are good styles to be familiar with). Overall, this makes JavaScript a useful second language to learn: programs will utilize similar structures, but demonstrate how code syntax can be abstracted in more general programming practices.

This chapter introduces the basic syntax of the JavaScript language, including variables, control structures, and functions. **Note** that this introduction assumes familiarity with Python, and introduces concepts in contrast to that languages. For more general purpose introductions, see the below resources.

17.1   Programming with JavaScript

Like Python, JavaScript is a **high-level, general-purpose, interpreted programming language**. The biggest difference is where it is most commonly used: while the Python interpreter is installed on a computer and is usually accessed through the command line (or through the Jupyter program), JavaScript interpreters are most commonly built into web browsers (such as Chrome or Firefox). Browsers are able to download scripts written in JavaScript, executing them line-by-line and use those instructions to manipulate what content is displayed.

It is also possible to execute JavaScript code via the command-line by using Node.js, allowing JavaScript to be a fully general language. However, JavaScript

is still most commonly used inside the browser, which is how it will be discussed in this book.

17.1.1   History and Versions

The JavaScript language was developed by Brendan Eich (the co-founder of Mozilla) in 1995 while he was working for Netscape. The original prototype of the language was created in 10 days… a fact which may help explain some of the quirks in the language.

Despite the names, *JavaScript* and the *Java* language have nothing to do with one another (and are in fact totally separate programming languages used in drastically different contexts). JavaScript was named after Java as a marketing ploy to cash in on the popularity of the latter.

Like Python, JavaScript continues to be developed through multiple versions, though *unlike* Python these versions are all backwards compatible. Each new version of JavaScript includes additional syntax shortcuts, specialized keywords, and additional core functions. The main limitation on utilizing new JavaScript features is whether the *interpreters* found in web browsers are able to support them. This module will use syntax and styles based on `ES5` (JavaScript 5), which was introduced in 2011 and is supported by all modern browsers (i.e., IE 10 or later).

- `ES6` was introduced in 2015 and provides features that work on many browsers, while `ES7` was finalized in 2016 but is not reliably supported

- Technically, JavaScript is an *implementation* of the the ECMAScript specification (which defines the structure and behaviors of multiple programming languages). Hence the `ES` in the version names.

17.1.2   Running JavaScript

JavaScript scripts are executed in a web browser as part of the browser rendering a web page. Since browsers render HTML content (in `.html` files), JavaScript scripts are specified within that HTML by using a `<script>` tag and specifying the *relative* path to the script file (usually a **.js** file) to execute. When the HTML being rendered (reading top to bottom) gets to that tag, the browser will download and execute the specified script file using the JavaScript interpreter:

```
<script src="path/to/my/script.js"></script>
```

- The `<script>` tag can be included anywhere in an HTML page. Most commonly it is either placed in the `<head>` in order for the script to be executed *before* the page content loads, or at the very end of the `<body>` in order for the script to be executed *after* the page content loads (and so can interact with that content).

- **IMPORTANT:** note that if you edit the `.js` script file, you will need to **reload** the page so that the browser can execute the script again (starting from the beginning, as if the page were viewed for the first time).

A webpage can include multiple `<script>` tags, each specifying their own script file. These scripts will be executed by the interpreter whenever they are encountered, top to bottom. And since variables and functions are usually defined *globally*, this means that any variables or functions created in one script will be available for use in the next (just like how variables created in one Jupyter cell are available in the next).

Thus additional JavaScript modules can be "imported" by including their script file as a `<script>` tag *before* the script that needs to use them (no explicit `import` statement is necessary). These scripts will then make additional **global variables** available for use by later scripts.

- Moreover, rather than downloading third-party modules as part of a package like Anaconda, the path to third-party JavaScript modules are specified as an internet URI so that the module is downloaded along with the rest of the web page. For example:

  ```
  <script src="https://d3js.org/d3.v4.min.js"></script>
  ```

  will "import" (download and include) the D3 library and provide a global `d3` variable for later scripts to use—similar to `import d3` in Python.

While JavaScript most commonly is used to manipulate the web page content and is thus pretty obvious to the user, it *also* can produce "terminal-like" output—including printed text and **error messages**. This output can be viewed through the **JavaScript Console** included as a *developer tool* in the Chrome browser (other browsers include similar tools):
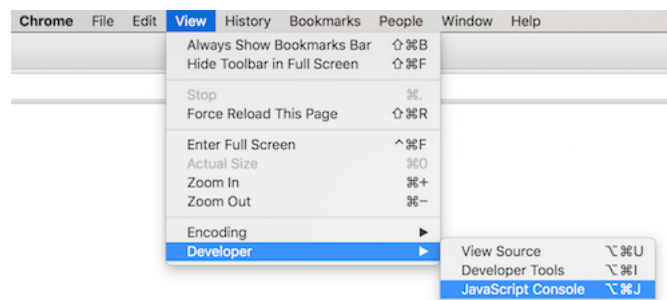

Figure 17.1: Accessing the developer console in Chrome.

Important: You should *always* have the JavaScript Console open when developing JavaScript code, since this is where any error messages will appear!

Finally, while a web browser is able to open any local `.html` file and run its

included `.js` scripts, certain interactions (e.g., downloading data from the internet via an HTTP request) may require a web page to be provided from a *web server* via the `http://` protocol, rather than the `file://` protocol. Luckily Python 3 provides a simple web server through the `http.server` module which you can use to "serve" the files to yourself:

```
# on the command-line:
cd path/to/project
python -m http.server
```

The served webpage will be available at http://localhost:8000/ ("localhost" means "this machine").

- Be sure to `cd` to the directory that contains the `index.html` file—otherwise you will be serving from a different folder so will need to adjust the URL's path.
- The `-m` option mean to run the built-in module as script, instead of just importing it.
- I highly recommend you run a local web server whenever you are doing web development!

17.2   JavaScript Basics

JavaScript scripts (`.js` files) have a similar overall structure to Python scripts: these files contain lines of code, each of which is a **statement** (instruction). The JavaScript interpreter executes each statement one at a time, top to bottom (responding to control structures and function calls of course).

Like Python, JavaScript code can include **comments**, which is text that is *ignored* by the interpreter. JavaScript includes two kinds of comments:

- Anything written after two slashes (`//`) until the end of the line is treated as an *inline comment* and ignored.
- Anything written between `/*` and `*/` is treated as a *block comment* and ignored. Block comments can span multiple lines.

```
/* This is a block comment
   It can span multiple lines */
console.log('Hello world!');  //this is an inline comment (like # in Python)
```

The above example code demonstrates the **console.log()** function, which is JavaScript's equivalent to `print()`—the output will be shown in the JavaScript console (in the Developer Tools). Thus we talk about "logging out" values in JavaScript, instead of "printing" values in Python.

- The `console.log()` function is technically a `log()` method belonging being called on a global `console` object.

Also notice that the example statement ends in a semicolon (**;**). All JavaScript statements end in semicolons, marking that statement as being finished (like a period in an English sentence). Unlike in Python, statements that do not end in semicolons can be allowed to continue onto the next line.

- Note that JavaScript tries to helpful and will often assume that statements end at the end of a line if the next line "looks like" a new statement. However, it occasionally screws up—and so best practice as a developer is to **always include the semicolons**.

17.2.1 Strict Mode

ES5 includes the ability for JavaScript to be interpreted in **strict mode**. Strict mode is more "strict" about how the interpreter understands the syntax: it is less likely to assume that certain programmer mistakes were intentional (and so try to run the code anyway). For example, in *strict mode* the interpreter will produce an *Error* if you try and use a variable that has not yet been defined, while without strict mode the code will just use an `undefined` value. Thus working in strict mode can help catch a lot of silly mistakes.

You declare that a script or function should be executed in strict mode by putting an interpreter declaration at the top of the file:

```
'use strict';
```

This is not a String, but rather a *declaration* to the interpreter about how it should be behave.

ALWAYS USE STRICT MODE! It will help avoid typo-based bugs, as well as enable your code to run more efficiently.

17.3 Variables

JavaScript variables are **declared** using the `var` keyword. This is like creating the "nametag"; once it has been declared, new values can be assigned to that variable without further declaration. Declared variables have a default value of `undefined`—a value representing that the variable has no value (similar to `None` in Python).

```javascript
var x = 4;  //declare and assign value
var y = x;  //declare and assign value
x = 5;      //assign value to existing variable

console.log(x+', '+y);  //5, 4
```

```javascript
var z;  //declare variable (not assigned)
console.log(z);  //undefined
```

JavaScript variables are conventionally named using camelCase: each word in the variable name is put together (without a separating _), with the first letter in subsequent words capitalized:

```javascript
var myVariable = 598;
var numberOfDaysInAYear = 365.25;
```

ES6 introduces the keyword **let** for declaring variables (to use instead of `var`). This works in a similar fashion, except that variables declared using `let` are **"block scoped"**, meaning that they are only available within the block in which they are defined, not the entire function. This is different than how Python works, but similar to how Java and C work. The `let` keyword is available in all modern browsers, and is increasingly common in JavaScript examples and tutorials. For this purposes of this book, you can view it as an interchangeable option with `var`.

17.3.1   Basic Data Types

JavaScript supports the similar basic data types as Python:

- **Numbers** are used to represent numeric data (JavaScript does not distinguish between integers and floats). Numbers support most of the same *mathematical* and operators as Python (you can't use `**` to raise to an exponent, and `//` is a comment not integer division). Common mathematical functions can be accessed through in the built-in `Math` global.

  ```javascript
  var x = 5;
  typeof x;  //'number'
  var y = x/4;
  typeof y;  //'number'

  //Numbers use floating point division
  console.log(x/4);  //1.25

  //Use the Math.floor() function to do integer division
  console.log( Math.floor(x/4) );  //1

  //Other common Math functions are available as well
  console.log(Math.sqrt(x));  //2.23606797749979
  ```

- **Strings** can be written in either single quotes (`'`) or double quotes (`"`), but most style guidelines recommend single quotes. Strings can be concatenated (but not multiplied!)

```
var name = 'Joel';
var greeting = 'Hello, my name is '+name; //concatenation
```

Strings also support many methods for working with them. Note that like Python, JavaScript strings are *immutable*, so most string methods return a new, altered string.

- **Booleans** in JavaScript are written in lowercase letters: `true` and `false`. Booleans can be produced using the same *relational operators* as in Python. However, the *logical operators* used on booleans are written using symbols rather than words: two *ampersands* (**&&**) mean "and", two *pipes* (**||**) mean "or", and an exclamation point (**!**) means "not":

```
//conjunction
boolOne && boolTwo //and (two ampersands)

//disjunction
boolOne || boolTwo //or (two pipes)

//negation
!boolOne //not (exclamation point)

//combining
P || !Q        //P or not Q
!P && Q        //not P and also Q
!(P && Q)      //not (P and Q both)
(!P) || (!Q)   //not P or not Q

3 < x && x < 5  //not as cool as Python
```

17.3.2 Type Coercion

Unlike Python, JavaScript will not throw errors if you try to apply operators (such as + or <) to different types. Instead, the interpreter will try to be "helpful" and **coerce** (convert) a value from one data type into another. While this means you don't have to explicitly use `str()` in order to print numbers, JavaScript's type coercion can produce a few quirks:

```
var x = '40' + 2;
console.log(x);  //=> '402'; the 2 is coerced to a String
var y = '40' - 4;
console.log(y);  //=> 36; can't subtract strings so '40' is coerced to a Number!
```

JavaScript will also attempt to coerce values when checking for equality with `==`:

```javascript
var num = 10
var str = '10'

console.log(num == str) //true, the values can be coerced into one another
```

In this case, the interpreter will coerce the Number `10` into the String `'10'` (since numbers can always be made into Strings), and since those Strings are the same, determines that the variables are equal.

In general this type of automatic coercion can lead to subtle bugs. Thus you should instead always use the `===` operator (and it's partner `!==`), which checks both value *and* type for equality:

```javascript
var num = 10
var str = '10'

console.log(num === str) //false, the values have different types
```

JavaScript will do its best to coerce any value when compared. Often this means converting values to Strings, but it will also commonly convert values into *booleans* to compare them. So for example:

```javascript
//compare an empty String to the number 0
console.log( '' == 0 ); //true; both can be coerced to a `false` value
```

This is because both the empty string `''` and `0` are considered **"falsey"** values (values that can be coerced to `false`). Other falsy values include `undefined`, `null`, and `NaN` (not a number). All other values will be coerced to `true`.

17.3.3 Arrays

Arrays are *ordered, one-dimensional sequences of values*—very similar to Python lists. They are written as literals inside square brackets `[]`. Individual elements can be accessed by (0-based) *index* using **bracket notation**.

```javascript
var names = ['John', 'Paul', 'George', 'Ringo'];
var letters = ['a', 'b', 'c'];
var numbers = [1, 2, 3];
var things = ['raindrops', 2.5, true, [5, 9, 8]];
var empty = [];

console.log( names[1] );  // "Paul"
console.log( things[3][2] );  // 8

letters[0] = 'z';
console.log( letters );  // ['z', 'b', 'c']
```

Note that it is possible to assign a value to *any* index in the array, even if that index is "out of bounds". This will *grow* the array (increase its length) to include that index—intermediate indices will be given values of undefined. The *length* of the array (accessed via the .length attribute) will always be the index of the "last" element + 1, even if there are fewer defined values within the array.

```javascript
var letters = ['a', 'b', 'c'];
console.log(letters.length);  // 3
letters[5] = 'f';  //grows the array
console.log(letters);  // [ 'a', 'b', 'c', , , 'f' ]
                       //blank spaces are undefined
console.log(letters.length);  // 6
```

Arrays also support a variety of methods that can be used to easily modify their elements. Common *functional programming*-style methods are described below.

17.3.4 Objects

Objects are *unordered, one-dimensional sequences of **key-value pairs**—*very similar to Python dictionaries. They are written as literals inside curly braces **{}**, with keys and values separated by colons **:**. Note that in JavaScript, string keys do *not* need to be written in quotes (the quotes are implied—the keys are in fact strings).

```javascript
//an object of ages (explicit Strings for keys)
//The `ages` object has a `sarah` property (with a value of 42)
var ages = {'sarah':42, 'amit':35, 'zhang':13};

//different properties can have the same values
//property names with non-letter characters must be in quotes
var meals = {breakfast:'coffee', lunch: 'coffee', 'afternoon tea': 'coffee'}

//values can be of different types (including arrays or other objects!)
var typeExamples = {number:12, string:'dog', array:[1,2,3]};

//objects can be empty (contains no properties)
var empty = {}
```

Object elements are also known as **properties**. For example, we say that the ages object has a sarah property (with a value of 42).

Object values can be access via **bracket** notation, specifying the *key* as the index. If a key does not have an explicit value associated with it, accessing that key produces undefined (the key's value is undefined).

```javascript
var favorites = {music: 'jazz', food: 'pizza', numbers: [12, 42]};
```

```javascript
//access variable
console.log( favorites['music'] ); //'jazz'

//assign variable
favorites['food'] = 'cake';
console.log( favorites['food'] ); //'cake'

//access undefined key
console.log( favorites['language'] ); //undefined
favorites['language'] = 'javascript'; //assign new key and value

//access nested values
console.log( favorites['numbers'][0] ); //12
```

Additionally, object values can also be accessed via **dot notation**, as if the properties were *attributes* of a class. This is often simpler to write and to read: remember to read the **.** as an `'s`!

```javascript
var favorites = {music: 'jazz', food: 'pizza', numbers: [12, 42]};

//access variable
console.log( favorites.music ); //'jazz'

//assign variable
favorites.food = 'cake';
console.log( favorites.food ); //'cake'

//access undefined key
console.log( favorites.language ); //undefined
favorites.language = 'javascript'; //assign new key and value

//access nested values
console.log( favorites.numbers[0] ); //12
```

- The one advantage to using *bracket notation* is that you can specify property names as variables or the results of an expression. Thus the recommendation is to use *dot notation* unless the property you wish to access is dynamically determined.

It is possible to get *arrays* of the object's keys calling the `Object.keys()` method and passing in the object you wish to get the keys of. Note that an equivalent function for values is not supported by most browsers.

```javascript
var ages = {sarah:42, amit:35, zhang:13};
var keys = Object.keys(ages); // [ 'sarah', 'amit', 'zhang' ]
```

17.4 Control Structures

JavaScript supports **control structures** (conditional `if` statements, `while` and `for` loops) in the same way that Python does, just with a slightly different syntax.

17.4.1 Conditionals

A JavaScript **if statement** is written using the following syntax:

```javascript
if(condition){
  //statements
}
else if(alternativeCondition) {
  //statements
}
else {
  //statements
}
```

In JavaScript, a **block** of statements (e.g., to conditionally execute) is written inside curly braces **{}**. JavaScript doesn't use indentation to designate blocks, though you should still indent blocks for readability. - As in Python, the `else if` and `else` clauses are optional.

Similar to Python, the **condition** can be any expression that evaluates to a Boolean value, and is placed inside parentheses. But since any value can be *coerced* into Booleans, you can put any value you want inside the `if` condition. This is actually really useful—since `undefined` is a falsy value, you can use this coercion to check if a variable has been assigned or not:

```javascript
//check if a `person` variable has a `name` property
if(person.name){
    console.log('Person does have a name!');
}
```

In the above example, the condition will only coerce to `true` if `person.name` is defined (*not* `undefined`) and is not empty. If somehow the variable has not been assigned (e.g., the user didn't fill out the form completely), then the condition will not be true.

Overall, while the syntax may be different, the control flow impacts are the same as in Python: conditional statements can be nested, and subsequent `if` statements are not necessarily mutually exclusive!

17.4.2   Loops

JavaScript supports **while loops** just like Python, with the syntactic change that conditions are listed in parentheses and blocks are written in curly braces:

```javascript
var count = 5;
while(count > 0){
   console.log(count);
   count = count - 1;
}
console.log("Blastoff!");
```

JavaScript also provides a **for loop** used for *definite iteration*, such as when you know how many times you want to continue through the loop. To understand the JavaScript `for` loop, consider the below loop for iterating through an array:

```javascript
var i = 0;
while(i < array.length){
   console.log(array[i]);
   i++; //shortcut for i = i+1
}
```

This loop has somewhat generic parts:

1. It initializes the *loop control variable* (`var i=0`)
2. It checks the condition each time through the loop (`i < array.length`)
3. It *increments* the loop control variable at the end of the block (`i++`). The `++` is a short cut for "add 1 and reassign" (since statements such as `i=i+1` are so common)

The JavaScript `for` loop combines these three steps into a single statement:

```javascript
for(var i=0; i<array.length; i++){
  console.log(array[i]);
}
```

Thus JavaScript makes it easy to keep track of a loop control variable (a counter), though you need to write out the initialization, condition, and update yourself.

- This is equivalent to the Python `for i in range(len(array)):`

Warning JavaScript *does* have a `for ... in` syntax. However, it doesn't work as you would expect for arrays (it iterates over "enumerable properties" rather than the specific indices), and so should **not** be used with arrays. Instead, `ES6` introduces a `for ... of` syntax for iterating over arrays, though it is not supported by all browsers. Thus the current best practice is to use the above `for` loop, or better yet the `forEach()` method described below.

If you need to iterate over the keys of an object, use the `Object.keys()` method to get an array to loop through!

17.5 Functions

And of course, JavaScript includes **functions** (named sequences of statements used to *abstract* code) just like Python. JavaScript functions are written using the following syntax:

```javascript
//A function named `makeFullName` that takes two arguments
//and returns the "full name" made from them
function makeFullName(firstName, lastName) {
  //Function body: perform tasks in here
  var fullName = firsName + " " + lastName;

  // Return: what you want the function to output
  return fullName;
}

// Call the makeFullName function with the values "Alice" and "Kim"
myName = makeFullName("Alice", "Kim")  // "Alice Kim"
```

Things to note about this syntax:

- Functions are defined by using the **function** keyword (placed before the name of the function). This acts similar to `def`.

- As with control structures, the **block** that is the function's *body* is written in curly braces **{}**.

Like Python, JavaScript variables are *scoped* to functions: any variables declared within a function will not be available outside of it. Functions are able to access, modify, and assign variables that are at a *higher* scope (e.g., global variables). This includes the arguments, which are implicitly declared *local* variables!

JavaScript does not support named keyword arguments (before `ES6` anyway). However, in JavaScript **all arguments are optional**. Any argument that is *not* (positionally) passed a specific value will be `undefined`. Any passed in value that does not have a variable defined for its position will not be assigned to a variable.

```javascript
function sayHello(name) {
    return "Hello, "+name;
}

//expected: argument is assigned a value
sayHello("Joel");  //"Hello, Joel"

//argument not assigned value (left undefined)
sayHello();  //"Hello, undefined"
```

```
//extra arguments (values) are not assigned to variables, so are ignored
sayHello("Joel","y'all");  //"Hello, Joel"
```

- If a function has an argument, that doesn't mean it got a value. If a function lacks an argument, that doesn't mean it wasn't given a value.

17.5.1   Functional Programming

JavaScript functions ARE values. This means that, as in Python, each function you define is in fact an object that can be assigned to other variables or passed around to other functions.

```
function sayHello(name){
  console.log("Hello, "+name);
}

//what kind of thing is `sayHello` ?
typeof sayHello;  //'function'

//assign the `sayHello` value to a new variable `greet`
var greet = sayHello;

//call the function assigned to the `greet` variable
greet("world");  //prints "Hello world"
```

Just like arrays and objects can be written as literals which can be *anonymously* passed into functions, JavaScript supports **anonymous functions** (similar to Python's anonymous lambdas):

```
//Example: an anonymous function (no name!)
//(We can't reference this without a name, so writing an anonymous function is
//not a valid statement)
function(person) {
   console.log("Hello, "+person);
}

//Anonymous function (value) assigned to variable
//equivalent to the version in the previous example
var sayHello = function(person) {
   console.log("Hello, "+person);
}
```

You can think of this structure as equivalent to declaring and assigning an array `var myVar = [1,2,3]`... just in this case instead of taking the anonymous array (right-hand side) and giving it a name, you're taking an *anonymous function* and giving it a name!

Using **anonymous callback functions** are incredibly important in JavaScript programming. Many common built-in functions use callback functions to specify some type of behavior, and these callbacks are normally specified as anonymous functions (rather than clutter the namespace).

17.5.1.1 Array Loops

As a common example, **arrays** support a variety of methods for performing *functional looping*—in fact, this is the recommended way to loop through an array!

We loop through an array by calling the [forEach(https://developer.mozilla.org/en-US/docs/Web/JavaScrip function on that array. This function takes in a *callback function* specifying what block of code to execute "for each" element:

```javascript
var array = ['a','b','c'];  //an array
var printItem = function(item) {  //something to do to an array item
   console.log(item);
}

array.forEach(printItem);  //do the "printItem" thing to EACH array item

//it is much more common to use anonymous function
array.forEach(function(item) {
   console.log(item);
});
```

- This is the (admittedly verbose) equivalent to Python's `for ... in` syntax.

- In the last line, we basically take the anonymous function assigned to `printItem` and "copy-paste" that literal definition as the argument to `forEach()`, instead of referring to the value by its name. This statement can be read as "take the `array` and `forEach` thing run this `function` on the `item`".

- Be careful about the closing braces and parentheses at the end of the anonymous function version: the brace is ending the anonymous function's block (and so ending that value), and the parenthesis is closing the argument list of `forEach()`!

- Fun fact: ES6 provides a syntactical shortcut for specifying anonymous functions that is less verbose called arrow functions.

Arrays also provide `map()`, `filter()` and `reduce()` functions (similar to those in Python) that can be used to do functional-style programming and mirror the functionality provided by Python list comprehensions:

```javascript
var array = [3,1,4,2,5];

//map: return a new array of transformed items
array.map(function(item) {
   return item*2;   //transform each item into its double
});  // [6, 2, 8, 4, 10]

//filter: return a new array of items that meet the criteria
array.filter(function(item){
   return item >= 3;  //keep if large
});  // [3, 4, 5]

//reduce: aggregate array elements into a single value
//reduce() also takes a second argument which is the starting value
array.reduce(function(runningTotal, item){
  var newTotal = runningTotal + item;
  return newTotal;
}, 0);
```

This type of functional programming is incredibly important when developing JavaScript programs (and especially using it for interactive applications)!

Resources

As the language used for web programming, JavaScript may have more freely available online learning resources than any other programming language! Some of my favorites include:

- A Re-Introduction to JavaScript a focused tutorial on the primariy language features
- You Don't Know JS a free textbook covering all aspects of the JavaScript language. Very readable and thorough, with lots of good examples.
- JavaScript for Cats a gentler introduction for "Scaredy-Cats"
- w3Schools JavaScript Reference a super-friendly reference for the language
- MDN JavaScript Reference a complete documentation of JavaScript, including tutorials
- Client Side Web Development a more extensive treatment of JavaScript; see Chapters 10-15.

Chapter 18

Web Programming

The most common use of programming in JavaScript is to create **dynamic, interactive web pages**: websites where the content changes over time or in response to user input *without the page reloading*. Almost any interaction that isn't following a hyperlink to a new web page is produce through JavaScript. This chapter provide an overview of using JavaScript to produce interactive web pages: using HTML syntax to define content, programmatically manipulating that content (using the D3 library), responding to user input, and dynamically downloading data.

18.1   Web Pages and HTML

A webpage is simply a set of files that the browser *renders* (shows) in a particular way, allowing you to interact with it. Webpage content (words, images, etc.) is formatted and structured using **HTML** (**H**yper**T**ext **M**arkup **L**anguage). This is code that functions similarly to Markdown by allowing you to *annotate* plain text (e.g., "this content is a heading", "this content is a hyperlink"), except it is much more flexible and powerful.

HTML annotates content by *surrounding* it with **tags**:

The **opening/start tag** comes before the content and tell the computer "I'm about to give you content with some meaning", while the **closing/end tag** comes after the content to tell the computer "I'm done giving content with that meaning." For example, the `<h1>` tag represents a top-level heading (equivalent to one `#` in Markdown), and so the open tag says "here's the start of the heading" and the closing tag says "that's the end of the heading". Taken together, the tags and the content they *contain* are called an **HTML Element**. A website is made of a bunch of these elements.

Figure 18.1: Element diagram

Tags are written with a less-than symbol `<`, then the name of the tag (often a single letter), then a greater-than symbol `>`. An *end tag* is written just like a *start tag*, but includes a forward slash `/` immediately after the less-than symbol—this indicates that the tag is closing the annotation. Line breaks and white space around tags (including indentation) is ignored.

- HTML tag names are not case sensitive, but you should always write them in all lowercase.
- Line breaks and white space around tags (including indentation) is ignored. Tags may thus be written on their own line, or inline with the content.

The *start tag* of an element may also contain one or more **attributes**. These are similar to attributes in object-oriented programming: they specify *properties*, options, or otherwise add additional meaning to an element. Like named keyword parameters in Python or HTTP query parameters, attributes are written in the format `attributeName=value` (no spaces are allowed around the `=`); values of attributes are almost always strings, and so are written in quotes. Multiple attributes are separated by spaces:

```
<tag attributeA="value" attributeB="value">
   content
</tag>
```

For example, a hyperlink anchor (`<a>`) uses a `href` ("**h**ypertext **ref**erence") attribute to specify where the content should link to:

```
<a href="https://ischool.uw.edu">iSchool homepage</a>
```

- In a hyperlink, the *content* of the tag is the displayed text, and the *attribute* specifies the link's URL. Contrast this to the same link in Markdown:

  ```
  [iSchool homepage](https://ischool.uw.edu)
  ```

Every HTML element can include an **id** attribute, which is used to give them a unique identifier so that we can refer to them later (e.g., from JavaScript). `id` attributes are named like variable names, and must be **unique** on the page.

```
<h1 id="title">My Web Page</h1>
```

Other common attributes include `style` for specifying appearance style rules, or `class` for giving a (space-separated) list of Cascading Style Sheet classifications.

Some HTML elements don't require a closing tag because they *can't* contain any content. These tags are often used for inserting media into a web page, such as with the `<img>` tag. With an `<img>` tag, you can specify the path to the image file in the `src` attribute, but the image element itself can't contain additional text or other content. Since it can't contain any content, we leave off the end tag entirely. However, we can optionally include the "end" slash `/` just before the greater-than symbol to indicate that the element is complete and to expect no further content:

```
<!-- images have no text content! -->
<img src="picture.png" alt="description of image for screen readers and indexers" />
```

18.1.1 Elements are Nested

HTML elements can be **nested**: that is, the content of an HTML element can contain *other* HTML tags (and thus other HTML elements):

Figure 18.2: An example of element nesting: the `<em>` element is nested in the `<h1>` element's content.

The semantic meaning indicated by an elements applies to *all* its content: thus all the text in the above example is a top-level heading, and the content "(with emphasis)" is emphasized as well.

Because elements can contain elements which can *themselves* contain elements, an HTML document ends up being structured as a **"tree"** of elements:

In an HTML document, the `<html>` element is the "root" element of the tree. Inside this we put a `<head>` element to represent "header" information (metadata that is **not**shown on the page itself), and a `<body>` element to contain the document's "body" (the shown content):

```
<html>
  <head>
    <title>The title that appears in the browser tab</title>
```
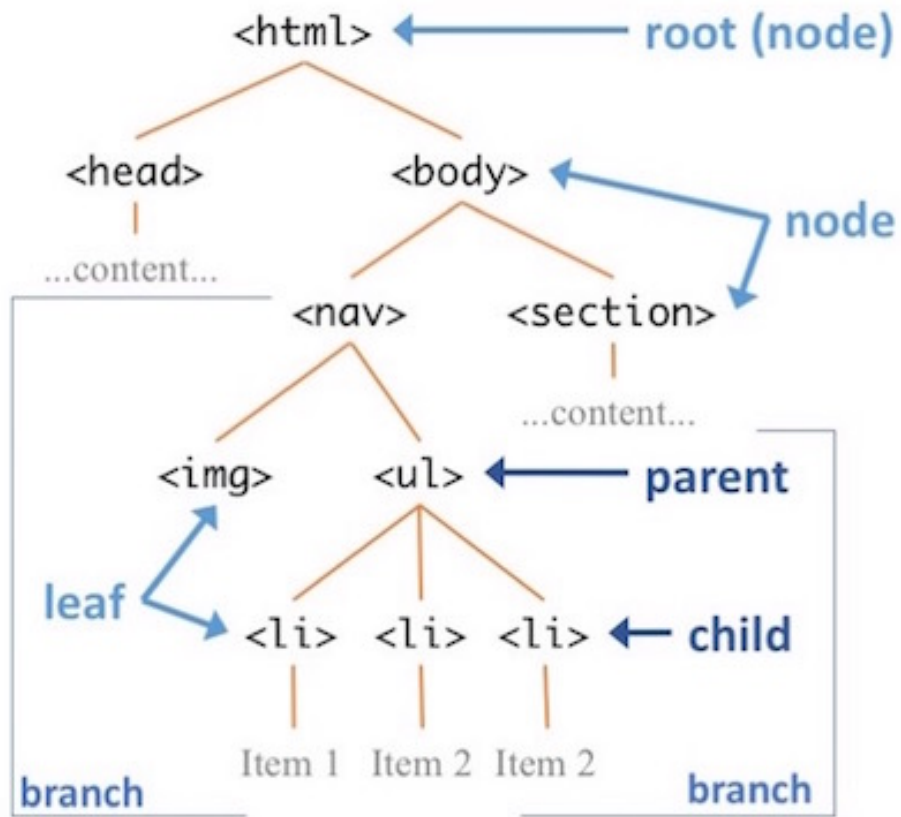
Figure 18.3: An example DOM tree (a tree of HTML elements).

```html
  </head>
  <body>
    <h1>Hello world!</h1>
    <p>This is <em>conteeeeent</em>!</p>
  <body>
</html>
```

This model of HTML as a tree of "nodes"—along with an API (programming interface) for manipulating them— is known as the **Document Object Model (DOM)**.

18.1.2   Scalable Vector Graphics (SVG)

Scalable Vector Graphics (SVG) is a syntax for representing ("coding") simple graphical images. For example, you can use SVG to define "a 100x200 pixel green rectangle at the top of the screen", or a squiggly line that follows a path a la Turtle Graphics! SVG images can be very complex and made up of lots of different shapes: most logos, icons, and other "non-photograph" images on the internet are defined using SVG.

SVG is used to represent vector graphics—that is, pictures defined in terms of the connection between points (lines and other shapes), rather than in terms of the pixels being displayed. In order to be shown on a screen, these lines are converted (*rastered*) into grids of pixels based on the size and resolution of the display. This allows SVG images to "scale" or "zoom" independent of the size of the display or the image—you never need to worry about things getting blurry as you make them larger!

SVG images are written in a XML (**Ex**tensible **M**arkup **L**anguage) syntax that is identical to that used by HTML (just with different "tags"). And in HTML5 (supported by all modern browsers), SVG elements can be included *directly* in the HTML, allowing you to include—and more importantly, manipulate—these elements as if they were any other HTML element.

An SVG image is specified with an **`<svg>`** tag, inside of which is *nested* the individual shapes and components that should be shown. `<svg>` tags are usually given `width` and `height` attributes specifying their size (in screen pixels). Inside the `<svg>` tag are nested further elements, each representing a different shape:

```html
<svg width=400 height=400> <!-- the image -->
   <!-- a rectangle -->
   <rect x=20 y=130 width=50 height=80 />

   <!-- a circle -->
   <circle cx=150 cy=150 r=35 />
```

```
   <!-- some text -->
   <text x=100 y=100>SVG!</text>

   <!-- a turtle-style "path" of a triangle -->
   <path d="M150 0 L75 200 L225 200 Z" />
</svg>
```

Notice that each "shape" uses different attributes to define its appearance:

- A `<rect>` has an (x,y) position, as well as a width and height.

- A `<circle>` has a "**c**enter-x" and "**c**enter-y" position, as well as a **r**adius.

- A `<text>` includes the text to show as the element's content.

- A `<path>` defines a turtle-style path as a string of instructions. For example, M means "**m**ove to" the following (absolute) coordinates, L means "draw a **l**ine to" the following (absolute) coordinates, and the Z means to "close/end" the path by drawing a line back to the start. See the SVG specification for complete details.

  Note that SVG paths can be quite complex. The below path represents the UW logo!

  ```
  <path d="M 111.003,0 L 111.003,18.344 L 123.896,18.344 L 109.501,71.915 C 109.50
  ```

Additionally, all SVG shape elements support attributes that indicate the *appearance* of that shape. These include (among other options):

- **fill** indicates the color that the shape should be "filled in". This can be a named colors or RGB hex codes.

- **stroke** indicates the color of the shape's *outline*. This can be different than the `fill` color.

- **stroke-width** can be used to specify the width of the shape's outline (in pixels).

- **style** allows you to specify CSS style properties (in a format `property:value;`), and is can be used for some appearance aspects that are not supported by specific attributes. For example, this can be used to make a shape transparent using the `opacity` property:

  ```
  <!-- a circle that is 50% transparent -->
  <circle cx=100 cy=100 r=50 fill="red" style="opacity:0.5;" />
  ```

18.2   DOM Manipulation with D3

You can use JavaScript to make a web page's content **dynamic** (change over time) and **interactive** (responding to user input) by writing a script that will *modify* the HTML elements that are shown on the page: whether by changing their content, attributes, or even creating new elements!

Web browsers render HTML into a **DOM** tree that provides an API for performing these operations (e.g., by calling methods on the provided `document` global module). While this is a perfectly reasonable and effective way of making interactive web pages, this chapter will focus on how to use the D3 library to interact with the DOM. The D3 library provides more powerful and (slightly) easier-to-use functions for manipulating the HTML. Moreover, learning these functions now will make it easier to later use D3 for its primary purpose: creating interactive visualizations (described in the next chapter).

D3 is a third-party library, so as with Python modules you will need to "download" it and "import" it into your script. You do this by including a `<script>` tag with a *link* to the online version of the library **before** the `<script>` tag for your script:

```html
<script src="https://d3js.org/d3.v4.min.js"></script>
```

Note that this means you will need to be connected to the internet to utilize this library. If you need to write your program offline, you can instead download the library (the `d3.zip`) and load the included `d3.min.js` script by specifying a relative path to it.

18.2.1   Selecting Elements

In order to modify the web page's HTML elements from a JavaScript script, the first thing you will need to do is get a **reference** to the particular element you wish to modify: that is, create a *variable* whose value is an object corresponding to that element.

In D3, you can get this object by using the **d3.select()** function. This function takes as an argument a **string** specifying *which* element in the DOM should be selected. This string can take a few different formats:

- Specifying the element's **tag name** will select the *first* element on the page (from the top) with that tag:

  ```javascript
  var header = d3.select('h1'); //select the FRIST <h1> element
  ```

  Note that the angle brackets `<>` are *not* included in the selector string—just the tag name itself.

This format works great if you know there is only one element of a type on the page (e.g., if there is only one `<svg>` image on the page).

- You can select a particular element by its **id** attribute by using that `id` with a **#** immediately front of it:

```
var element = d3.select('#myElem'); //select element with attribute `id="myElem"
```

The # symbol can be read as "thing with id of". This format works great to select a particular element from many (as long as that element has an `id` attribute of course!)

- You can also use most any other standard CSS selector to be more precise about which element you wish to reference. In any case, the `d3.select()` function will return the **first** matching element.

The `d3.select()` function returns an object that is an instance of a **d3.selection** class. You can think of these as being "D3-specific" versions of that HTML DOM element. This object will provide a variety of methods that you can use to manipulate that element on the web page, as described below.

It is possible to select more than one element by using the `d3.selectAll()` function. This function also takes a selector string as an argument, but instead returns a `d3.selection` object that refers to **all** elements that "match" the selector. For example, `d3.selectAll('p')` will return an object representing all of the `<p>` (paragraph) elements on the page. This will allow you to easily manipulate large numbers of similar elements (e.g., all of the `<circle>` elements in a scatter plot).

- All the `d3.selection` object methods are **vectorized** (like with a `pandas` Series), meaning that they will apply to *each* selected DOM element. Thus calling a method to change the result of a `d3.selectAll()` function will change *all* of those elements in the same way. See below for examples.

It is also possible to call the `select()` and `selectAll()` methods on a `d3.selection` object. In this case, D3 will return the matching **child** element(s) from the DOM tree. For example, consider the HTML:

```
<div id="firstGroup">
  <p>First paragraph</p>
</div>
<div id="secondGroup">
  <p>Second paragraph</p>
  <p>Third paragraph</p>
</div>
```

In order to get a reference to the "Second paragraph", you could use the following:

```
var secondGroup = d3.select('#secondGroup');  //select second <div> by `id`
var paragraph = secondGroup.select(`p`);  //select first <p> inside that element
```

This allows you to interact with only particular "branches" of the DOM tree by first selecting the "root" of that branch, and then selecting the element(s) from that branch that you care about.

18.2.2 Changing Elements

Once you have a *reference* to an element, you can call methods on that object in order to modify its state in the DOM—which will in turn modify how it *currently* is displayed on the page. Thus by calling these methods, you are dynamically changing the web page's content!

Important: these methods *do not change the* `.html` *source code file!* Instead, they just change the *rendered DOM elements* (think: the content stored in the computer's memory rather than in a file). If you refresh the page, the content will go back to how the `.html` source code file specifies it should appear—with any loaded `.js` scripts being applied. With such dynamic content, what is shown on the page is the HTML with the JavaScript modifications added in, which may be a far cry from the `.html` content itself!

18.2.2.1 Changing Content

You can change a selected element's **content** (e.g., the stuff between the start and close tags) by using the `.text()` method. This method takes as an argument the new content for the element (which will replace any old content):

```
var paragraph = d3.select('p');
paragraph.text('This is new content!');  //change the paragraph's content.
```

Remember that an element's contents can include *other HTML elements*. However, any HTML included in the argument string will be *escaped* and shown as text content (e.g., showing the <>). For example:

```
var paragraph = d3.select('p');
paragraph.text('This is <em>new</em> content!');
```

will change the paragraph to display: "This is new content!"

If you want to include HTML elements inside the new content, you should instead use the `.html()` method:

```
var paragraph = d3.select('p');
paragraph.html('This is <em>new</em> content!');
```

will change the paragraph to display: "This is *new* content!"

- Be careful using `.html()` to display any user-provided data, since the content they give could include malicious `<script>` tags!
- The `.html()` method is primarily used for "inline" formatting tags like `<em>` (emphasis, italics) or `<strong>` (bold).
- The `.html()` method does not work with SVG elements; see below for other options.

Both the `.text()` and `.html()` methods can also be called without *any* arguments. In this situation, the methods *return* a string of the element's content without changing anything (`.text()` returns a string with all of the HTML tags removed). This lets you "get" the current contents of an element, such if if you want to do any string processing on it.

Calling either method on a selection with multiple elements (e.g., from `d3.selectAll()`) will cause the change to be made to *all* of the elements in the selection.

18.2.2.2 Changing Attributes

You can change a selected element's **attributes** by calling the **`.attr()`** method on that selection. This method takes two arguments: the name of the attribute to change, and the new value to give to that attribute:

```
var circle = d3.select('circle');  //select a circle
circle.attr('fill', 'green');  //make the circle green
```

The `.attr()` method *returns* a reference to the object that it modified. You can then use that returned value as the *anonymous subject* of additional method calls. This is called **chaining methods**:

```
var circle = d3.select('circle')  //no semicolon!
    .attr('fill', 'green')        //call `.attr()` on (anonymous) result
    .attr('stroke', 'yellow')     //call `.attr() on (anonymous) result
    .attr('stroke-width', 5);     //semi-colon when all done
```

The first `d3.select()` call returns a selection for a `<circle>` element, which we then call `.attr()` on to change its fill color. That method returns *the same* circle element, which we then call `.attr()` to change its stroke color. *That* method then returns the same circle element again, which call `.attr()` on a third time to change its stroke width. The third `.attr()` returns the same circle yet again, which is the ultimate result of this expression and so is assigned to the `circle` variable.

- Method chaining is very common when working with D3 and in JavaScript in general, such as when you want to do lots of things in sequence to the same set of objects.

The `.attr()` method can also be called without the second argument (the new value), in which case it returns the *current* value of the specified attribute.

Calling this method on a selection with multiple elements (e.g., from `d3.selectAll()`) will cause the change to be made to *all* of the elements in the selection.

In addition to changing the attributes, you can also change components of the CSS `style` attribute directly using the `.style()` method. This takes similar arguments, and can be used for easily adjusting the style.

18.2.3 Adding Elements

You can use JavaScript to add new elements as *children* of a selection (e.g., inside that element) by using the **append()** method. This method takes as an argument the *name of the tag* to create and add (without the `<>`). The `.append()` method *returns* a reference to the newly created element, allowing you to call further methods (e.g., `.text()`, `.attr()`) to specify the content or attributes of this new element.

For example, if you have HTML:

```html
<div id="content">
</div>
```

then you could add a new `<p>` *inside* that `<div>`, along with some content:

```javascript
var div = d3.select('#content');  //get reference to the <div>
var newP = div.append('p');  //add a new <p>
newP.text('Hello world!');  //set the content
```

which will produce:

```html
<div id="content">
   <p>Hello world!</p>
</div>
```

You can of course use *method chaining* here as well, including to add and create multiple new elements at once!

Note that `append()` adds an element as the "last" child of the parent. If you wish to insert an element *before* a sibling, using the `insert()` method with a second argument of a selector string for which element you want to the new one to precede. You can also remove a selected element by calling the `remove()` method on it.

Calling any of these methods on a selection with multiple elements (e.g., from `d3.selectAll()`) will cause the change to be made to *all* of the elements in the selection—each element will have a new child added, or each element will be removed.

18.3   User Events

In order to make your page **interactive** (that is, able to change in response to user actions), you need to be able to respond to *user events.* Whenever a user interacts with a computer, the operating system announces that interaction as an **event**—the *event* of a button being clicked, the *event* of the mouse being moved, the *event* of a keyboard key being pressed, etc. These events are **broadcast** to the entire system, allowing any application (including the browser) to "respond" the occurrence of the event, such as by executing a particular JavaScript function.

Thus in order to respond to user actions (and the *events* those actions generate), you need to define a function that will be executed **when the event occurs**. You will define a function as normal, but the function will not get called by you as a particular step or statement in your script. Instead, the function you specify will be executed *by the system* when an event occurs, which will be at some indeterminate time in the future. This process is known as event-driven programming. It is also an example of **asynchronous programming**: in which statements are not executed in a single order one after another ("synchronously"), but may occur "out of order" or even at the same time!

In order for your script to respond to user events, you need to *register an event listener.* This is a bit like following someone on social media: you specify that you want to "listen" for updates from that person, and what you want to do when you "hear" some news from that person.

You can register event listeners using D3 by calling the **on()** method on an selected element (indicating that you want to do something *on* hearing an event). This method takes two arguments: a string representing what kind of event you want to listen for, and a *callback function* to execute when you hear that event:

```
//a (named) callback function
function onClickCallback() {
    console.log("You clicked me!");
}

var button = d3.select('button'); //reference the element you want to "listen" to
button.on('click', onClickCallback);  //register a listener for 'click' events
```

When the button is clicked by the user, it will "shout" out a `'click'` event ("I was clicked! I was clicked!"). Because you have set up a "listener" (an alert/notification) for such an occurrence, your script will be able to do something—and that something that it will do is run the specified callback function.

- It's like you handed someone a recipe and told them "when I call you, follow this recipe to bake a cake!"

- Note that this listener *only* applies to that button—if you wanted to respond to a different button, you'd need to register a separate listener!

There are numerous different events you can listen for, including `'dblclick'` (double-click), `'keydown'` (keyboard key is pressed), `'hover'` (mouse "hovers" over the element), etc. See https://developer.mozilla.org/en-US/docs/Web/Events for a complete list of browser-supported events.

Note that it is *much* more common to use an *anonymous function* as the event callback:

```
var button = d3.select('button');
button.on('click', function() {
    //do something here
    console.log("You clicked me!");
});
```

If you want to know more details about the event (e.g., *where* the user clicked with the mouse, *which* button was clicked), you can access the `d3.event` variable from inside the callback, which is an object with properties about the *current* event.

Note that these event callback functions can occur repeatedly, over and over again (e.g., every time the user clicks the button). This makes them potentially act a bit like the body of a `while` loop. However, because these callbacks are *functions* any variables defined within them are **scoped** to that function, and will not be available on subsequent executions. Thus if you want to keep track of some additional information (e.g., how many times the button was clicked), you will need to use a **global** variable declared outside of the function. Such variables can be used to represent the **state** (situation) of the program, which can then be used to determine what behavior to perform when an event occurs, following the below pattern:

```
WHEN an event occurs {
    check the STATE of the program;
    DECIDE what to do based on that state;
    UPDATE the state as necessary for the next event;
}
```

For example:

```
var clickCount = 0;  //keep track of the "state"
d3.select('button').on('click', function() {
    if(clickCount > 10) {  //decide what to do
        console.log("I think you've had enough");
    }
    else {
        clickCount++;  //change state (+1)
        console.log('You clicked me!');
```

```
    }
});
```

"State" variables are often objects, with individual values stored as the properties. This provides a name-spacing feature, and helps to keep the code from being cluttered with too many variables.

18.4   AJAX

While normally HTTP requests for data are sent through the browser (by entering a URL in the address bar), it is also possible to use D3 to programmatically send HTTP requests, similar to what you did in Python.

Having JavaScript code send an HTTP request is known as sending an **AJAX** request. *AJAX* is an acronym for **A**sychronous **J**avaScript **A**nd **X**ML, and refers collectively to the techniques used to send these requests.

Fun fact: AJAX was developed by Microsoft in the late 1990s, and originally was used to send requests for XML data. However, in modern web programming most AJAX requests sent to web services are used to download JSON data instead. Nevertheless, the AJAX name has stuck (instead of being renamed "AJAJ").

The other important part of using AJAX requests is that they are **asynchronous**. Downloading data from the internet can take a while (depending on the speed of the web server, the strength of the internet connection, etc). Rather than having the program "pause" and wait for this download to finish (as with *synchronous* programming, like you did in Python), AJAX requests will occur *at the same time* that the rest of the code is being executed. Thus the download and the remaining script will not be synchronized; they will be "asynchronous".

- You call a function to send the AJAX request, but then your script *keeps going*—executing the next line of code while the request is processed by the server and the data downloads "in the background".

- In order to deal with the response whenever it arrives, you specify a *callback function*. This callback will be executed once the data has finished downloading (conceptually similar to "when the 'response received' event occurs"). This callback function will be *passed* the response's data for you to use.

In order to send AJAX requests (for JSON data) using D3, you can use the `d3.json()` function. This function takes two parameters: the URI to query, and a callback to be executed once the response is received:

```
var uri = 'https://api.github.com/search/repositories?q=d3&sort=stars';

//send query to uri, specify callback function
d3.json(uri, function(data) {
    console.log("Data recieved!");
    console.log(data);
});

console.log('request sent');  //this statement will execute before data is received!
```

The callback function should take a single argument: the response's content in JSON format (that is, as a JavaScript object). Note that this content is extracted from the response "envelope" for us automatically.

- Remember that the `d3.json()` function is *asynchronous*: the script will keep running (executing the next line) while the data downloads; the callback will be execute at an unspecified point in the future

In some browsers, you will not be able to send an AJAX request when the web page is loading via the `file://` protocol (e.g., by double-clicking on the `.html` file). This is a security feature to keep you from accidentally running an HTML file that contains malicious JavaScript that will download a virus onto your computer. Instead, you should run a local web server for testing AJAX requests, as described in the previous chapter.

D3 also provides methods for sending AJAX requests that handle different data formats automatically. For example, `d3.csv()` will fetch an array of objects where each object's "keys" are based on the first (header) row in the CSV file. See the d3.requests module for details.

Resources

- Client-Side Web Development (Joel Ross). An more extensive treatment of web programming; see HTML Fundamentals for HTML specifics. You may also be interested in the materials from INFX598 Web Development.
- What is the DOM?
- SVG Tutorial (MDN); also the tutorial from w3schools.
- D3 Selections
- An Introduction to AJAX

Chapter 19

D3 Visualizations

While the D3 library can be used to do fundamental DOM manipulations and event handling, the true power of the library comes in its ability to create **dynamic, data-driven visualizations**—graphical representations of data sets that can change in response to changes in the data. In particular, D3 provides robust tools that make it easy to relate (**join**) the *DOM elements* shown on the screen (most commonly, SVG shapes) to the values in a data *array*. Then it is just up to you to specify how the data should be related to *visual properties* of the DOM in order to create a visualization.

This chapter will introduce the basics of creating data-driven visualizations using using D3, including data joining, data scaling, axes and decorations, and animations.

19.1   The Data Join

To start, consider an **array of objects** representing a *table* of data (similar to what you've seen before):

```
var peopleTable = [
    {name: 'Ada', mathExam: 100, spanishExam: 83},
    {name: 'Bob', mathExam: 82, spanishExam: 88},
    {name: 'Chris', mathExam: 78, spanishExam: 92},
    {name: 'Diya', mathExam: 91, spanishExam: 79},
    {name: 'Emma', mathExam: 93, spanishExam: 87}
];
```

At its most basic level, a **data visualization** is a graphical image whose components (e.g., shapes) represent different elements in the array (observations/rows

of the data). For example, a bar chat may have a single `rectangle` for each element in the array, a scatter plot may have a single `circle` for each element, and a line chart may have a `path` with a different "control point" for each element. D3 will let you *programmatically* associate an SVG element (or set of elements) with each element in a JavaScript array in order to define this visualization. For example, each of the five items in the above `peopleTable` could be associated with one of five `<rect>` elements in a bar chart.

Moreover, you also want each shape in your visualization to have a different appearance based on the properties (*fields*) of the data. For example, the `width` attribute of a `<rect>` may depend on the `mathExam` of the array element, and the `y` attribute may depend on the *index* of that element in the array. Thus you need to define a **mapping** between data properties and the ***visual attributes*** (size, position, color, etc.) of each component in the visualization.

D3 lets you create a mapping between an *array* of data and a *selection* of DOM elements—what is called **the data join** (the "joining" of data with the selection). You establish this mapping by calling the **data()** method *on* the selection of DOM elements, and passing in the array of data you wish to map to those elements:

```
var rects = svg.selectAll('rects');  //selection of all rects in the SVG
var dataJoin = rects.data(peopleTable);  //join with the people table
```

This *join* associates each element in the array with a single element in the selection pair-wise (e.g., the first DOM element is associated with the first array element, the second with second, and so on).

Once data has been joined to the selection, it is possible to modify the attributes of the DOM elements (as you've done previously) utilizing that element's associated **datum** (data item). You do this by passing a *callback function* to the `attr()` method. This callback function can take in the *datum* and the *index* in the array as arguments, and should return the appropriate value for the attribute for that particular element:

```
//a function that determines width of a rectangle based on a datum
var calcWidth = function(datum, index){
    return datum.mathExam;  //width is the math score
}

//set the width attribute, using the callback to calculate the width
//for each (joined) element in the selection
dataJoin.attr('width', calcWidth);
```

- Remember that the `attr()` function is *vectorized*, so called individually on each element in the selection!

- Note that the `index` was not used—technically, you could have left it out of the argument list!

- If you wish to refer to the DOM element itself inside the callback function, that value is assigned to a local variable called **this**.

As you may have guessed, we almost always use *anonymous callback functions* for this, and abbreviate the arguments to d and i to save space:

```
//this is equivalent to the above
dataJoin.attr('width', function(d,i) { return d.mathExam; });  //compact one-line!
```

You can of course use this process to specify all of the different attributes of an SVG element, including sizes, positions, fill and stroke colors, etc. It is also possible to pass a calculating callback function to most selection methods (including e.g. the style() function if you want to use CSS styles).

- Similarly, the on() event handler callback takes in the *datum* and *index* as arguments, if you wish to refer to the joined data when a user clicks on an element!

19.1.1   Entering and Exiting Elements

Importantly, it is possible that the *size* of the data array might not match the size of the selection! While the data table might have 5 elements (as above), the selection might have 3, 6, or even 0 <rect> elements! In this situation, you cannot just join the the data to the selection, since there isn't a one-to-one mapping.

In D3, any joined *data* elements that lack corresponding DOM elements are referred to as the **enter selection** (they are data elements that need to "enter" the visualization). They can be accessed by calling the **enter()** method *on the data join*. This will return a selection of null-like "placeholder" elements, each of which is joined with a data item (that didn't otherwise have an element to join with). Most commonly, you then use the append() method to create a new DOM element for each item in this selection:

```
<!-- Assume an empty list -->
<ul id="#list"></ul>
```

```
//add a <li> for each number in an array:
var listItems = d3.select('#list').selectAll('li');  //select the (zero) <li> elements
var dataJoin = listItems.data([3, 1, 4, 1, 5]);  //join with an array of data
var enterSelection = dataJoin.enter();  //get the enter selection
enterSelection.append('li') //add a new <li> for each "placeholder"
    .text(function(d){ return d; })  //set the text to be the datum
```

This would produce the DOM tree:

```
<ul id="#list">
  <li>3</li>
  <li>1</li>
```

```
  <li>4</li>
  <li>1</li>
  <li>5</li>
</ul>
```

Note that we can and do chain most of these calls together, rather than defining a separate `enterSelection` variable.

Important! Notice how this lets you create elements for data without needing to specify any DOM for those elements ahead of time! This means that you do not need to write any SVG elements in the HTML, and instead can just `append()` elements for the `enter()`ing data.

Equivalently, any *DOM elements* that lack a joined data element are referred to as the **exit selection** (they are DOM elements that need to "exit" the visualization). They can be accessed by calling the **exit()** method *on the data join*. This will return a selection of elements, none of which have a datum joined to them. Most commonly, you then use `remove()` to remove these data-less elements from the DOM:

```
//select the above HTML (5 <li> elements)
var listItems = d3.select('#list').selectAll('li');  //select the (5) <li>
var dataJoin = listItems.data([3,4,5]);  //join with an (smaller) array of data
var exitSelection = dataJoin.exit();  //get the exit selection
exitSelection.remove() //remove the extraneous <li>
```

To summarize:

- `data()` returns the items in the data array *and* in the DOM
- `enter()` returns items in the data array *and not* in the DOM
- `exit()` return items in the DOM *and not* in the data array

19.1.1.1 Object Consistency

As data is "entering" and "exiting" over time (e.g., in response to user interaction), it is important to make sure that you are *consistent* about the mapping between the data and DOM elements. If you do a second data join with a smaller array of data (a sub-list), you want to make sure you remove the correct elements (and not just the ones at the end of the list). Similarly, if you do a data join with a larger array of data, you want to make sure you append elements only for the data that isn't already shown.

By default, the `data()` join will associate the first datum with the first DOM element, the second datum with the second DOM element, and so on. But this means that if you later remove the first datum element, then "which" DOM element is associated with which datum will change (e.g., the previously-second datum will be joined with the first DOM element). This can cause problems, particularly for utilizing animations.

In order to make sure that your joining remains consistent, you specify a second argument to the `data()` function, known as the **key function**. This is a *callback function* (which takes the datum and index as arguments, as usual) that should return a *unique string identifier* for each datum—in effect, the "key" that D3 can use to look up that datum when joining:

```
//rectangle selected from above example
var rects = svg.selectAll('rects');
var dataJoin = rects.data(peopleTable, function(d) {
    return d.name;  // use the `name` property as the "key"
});
```

It is common to use properties such as `name` or `id` as the key function—if you are familiar with databases, these would be the foreign keys.

Caution, when you first call the `data()` function, the key function callback is executed once for each DOM element already in the selection, *and then* once for each new datum in the data. This means that if your selection included any un-joined DOM elements (e.g., because you started off with some hard-coded `<rect>` elements), then the `datum` argument to the callback will be `undefined`. The best solution for this is to always `append()` entering DOM elements!

19.1.1.2 The General Update Pattern

When modifying the visualized data in response to user interaction, it is very common for a data join to include both "entering" and "existing" data (e.g., some new data added, and some old data removed). In fact, whenever you do a data join, there are 3 possible situations:

1. The data is *"updating"*, and the DOM attributes need to be modified accordingly. This data were previously joined to DOM elements, and so will be included in the `data()` selection.
2. The data is *"incoming"*, and so new elements need to be added to the DOM. This data will be included in the `enter()` selection.
3. The data is *"outgoing"*, and so elements need to be removed from the DOM. This data will be included in the `exit()` selection.

As such, "updating" a visualization with a new data join involves a few steps, as detailed below:

```
function update(newDataArray) {
    //perform the data join with the "new" data list
    var rects = svg.selectAll('rect')
                .data(newDataArray, keyFunc);  //use key function for consistency

    //Update already bound elements (that are not coming or going)
    rects.classed('updated', true);  //e.g., add style class to updating
```

```
    //Handle entering data
    var present = rects.enter().append('rect')  //add new DOM elements
               .classed('new', true)  //e.g., add style class to entering
               .merge(rects); //save new DOM elements in a selection

    //Handle now present data (the merged selection)
    present.classed('here', true);  //e.g., add style class to current (including ne

    //Handle exiting data (from original selection)
    rects.exit().remove();
}
```

The stages of this process are:

1. Use `data()` to join the elements; modify the attributes of any that are bound.
2. Use `enter()` to create new elements; modify the attributes of the new elements.
3. Use `merge()` to combine the old and new; modify the attributes of anything that will stay.
4. Use `exit()` to remove old elements.

The above example encapsulates these steps into a function for re-use!

This process is known as the *General Update Pattern*, and is the recommended way of handling *dynamic* (interactive) data visualizations. Every time the data needs to change—whether because the user interacted with the web page, or the the data was being "live streamed" from an API—you call this general update function. The function will join the now-current data to visualize, and then update the DOM elements that make up the visualization in order to match the latest data.

In developing the `update()` function, you should specify what the visualization should look like for *any* set of data. Then you can change the data however you want, and the visualization will continue to reflect the data!

19.2   Animation

D3 is primarily used for *dynamic* data visualizations, in which the data being represented may change over time (otherwise, you would just use Illustrator or a charting library). While the *General Update Pattern* allows you to easily implement consistent changes to the visualization when the data set change, you often want to add a "wow" factor by having those changes be **animated**. Elements should fly to their new position, slide into place, fade in or out, or perform some other kind of visual change over a short time. This can help

the user understand what is going on as the data changes, as well as make the visualization seem more "smooth".

With D3, you can animate element changes by using **transitions**. A transition applies changes to an element (e.g, `attr()` method calls) over a length of time, rather than instantaneously. For example, rather than instantly changing an rectangle's `width` attribute from `0` to `100`, you could have that attribute take 1 second to change, causing the rectangle to "grow".

D3 transitions use linear interpolation to determine the value of the attributes at different times. So if you transitioned the width from `0` to `100` over 1 second, at the start of the second the width would be `0`, at the end it would be `100`, and halfway through (at 0.5 seconds) it would be `50`.

You create a transition by calling the **transition()** method on a selection. This creates a new `transition`, upon which you can call normal DOM manipulation methods (specifically: `attr()`, `style()` and `text()`):

```
d3.selectAll('rect').attr('width', 10).attr('height', 10);  //start at size 10

d3.selectAll('rect').transition()  //create a transition
    .attr('width', 100)   //call attribute to change over time
    .attr('height', 100)  //this will animate simultaneously
```

This example will cause the rectangle to "grow"! Note that in practice, anything after a `transition()` call in a method chain will be animated.

By default, transitions occur over 250 milliseconds (0.25 seconds). However, you can call additional methods on the `transition` in order to adjust this duration:

```
d3.selectAll('rect').transition()  //create a transition
    .duration(1000)  //animation takes 1000ms (1 second)
    .delay(100)       //wait 100ms before starting
    .attr('width', 100)   //call attribute to change over time
    .attr('height', 100)  //this will animate simultaneously
```

You can use a sequence of `transition()` calls to make multiple transitions that occur one after another:

```
d3.selectAll('rect')
    .transition()  //create a transition
        .attr('width', 100)   //first change the width
    .transition()  //a second transition (occurs afterwards)
        .attr('height', 100)  //then change the height
```

Transitions are a fun addition to include with any dynamic visualization; just add the `transition()` call into the chain before you update the attributes! Indeed, in the *General Update Pattern*, transitions can be used for all three of entering, exiting, and updating elements. See this demo for an example.

19.3   Margins and Positioning

When implementing a visualization, you often want to position groups of elements together. For example, you may want all of the circles in a scatter plot to be "centered" in the `<svg>` image, with some white space on the side (a **margin**) to use for titles or axis labels. While it is possible to do this positioning by carefully specifying the attributes (e.g., add `30` to each element's `x` attribute to give 30 pixels of white space), that can become tedious and error-prone—particularly if you want to change the spacing later!

However, it is possible to do this easily using SVG (with or without D3)! You are able to **group** SVG elements together by nesting them inside of a `<g>` (**g**roup) element. A `<g>` acts somewhat like a `<div>` in HTML—it has no visual appearance on its own, but can help to semantically organize DOM elements. You can even give `<g>` elements `id` attributes, allowing you to easily select just the shapes inside of that group.

All elements in a `<g>` will be positioned **relative to the group**. A `<rect>` with an `x` of `0` will be positioned at the edge of its parent `<g>`—where-ever that `<g>` happens to be! Thus you can easily add margins to groups of elements by nesting them in a `<g>`, and then positioning that `<g>` correctly

The easiest way to position a `<g>` element is to specify it's **`transform`** attribute. This attribute takes as a value a string representing how that element should "move" (be transformed). The "transformation string" is written like a sequence of space-separated function calls, each specifying a different movement that should be applied. For example:

```
<g transform="translate(20,50)">  <!-- move 20 in x, 50 in y -->
   <rect x=0 y=0 width=20 height=20 fill="red"></rect>
   <rect x=0 y=30 width=20 height=20 fill="blue"></rect>
</g>
```

specifies that a group should be **translated** or moved by 20 units (pixels) in the x direction and 50 units in the y direction.

- Other transformations include `rotate(degrees)` to rotate an element *counterclockwise* by the given degrees. You can both translate and rotate (in that order) using e.g., `transform="translate(20,50) rotate(45)"`.

- All SVG elements support the `transform` attribute. In fact, angle can be used as a visual attribute!

In D3, you use `<g>` groups to do positioning by creating a group with the appropriate transformation (constructing the "transformation function string" using String concatenation), then appending elements to that group:

```
var group = svg.append('g') //add a group
    //move over to add margins (specified as variables)
    .attr('transform','translate('+marginLeft+','+marginTop+')')

//do the normal data joining
group.selectAll('rect').data(myData)  //select all the rectangles in that group
    .enter().append('rect')  //create new rectangles as normal
      .attr('x',0)  //will be relative to the <g>
      .attr('y',0)  //will be relative to the <g>
```

The margins are usually specified using variables. See this demo for an example (note: it uses D3 version 3).

19.4 Scales

In the `peopleTable` example, we *mapped* exam scores to the `width` attribute directly: each point on an exam corresponded to a single unit (pixel) of width. But what if you were visualizing very small data (e.g., daily interest on a small investment) or very large data (e.g., number of books held by a library)? You would need to *scale* the values used: that is, $0.001 earnings might be 20 pixels, or 100 books might be a single pixel.
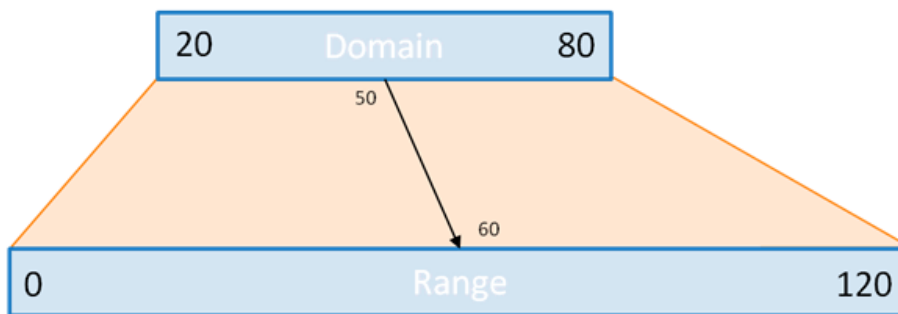

Figure 19.1: Scaling example

In D3, a **scale** is a function that *maps* from the data ***domain*** (in data values) to the visualization's ***range*** (in pixel values) in a consistent way. For example, if you wanted to perform the scaling illustrated in the above diagram you would need to have the *domain* of 20 to 80 (length of 60) map to the *range* of 0 to 120 (length of 120). You could write a function that does the math to do this for any individual value within the range!

```javascript
var diagramScale = function(value){
  var domainLength = 80-20; //length of domain
  var rangeLength = 120-0; //length of range

  //transform value to be between 0 and domainLength
  var shifted = value - 20;  //e.g., 50 => 30

  //scale (enlarge) the domain to the range
  var rangeValue = shifted*(rangeLength/domainLength);  //e.g., 30 => 60

  return rangeValue;
}

//example
var result = diagramScale(50);  //60, as above
```

This math is doable, but can be tedious—especially if the mapping needs to be more complex (such as using a *logarithmic* scale, or scaling values like colors).

Because scaling is such a common operation, D3 provides a set of helper functions that can be used to easily generate these *scaling functions*, allowing you to quickly specify a mapping (and dynamically change that mapping if the domain of the displayed data changes!)

You can create a simple (linear) scale by calling the `d3.scaleLinear()` function. This function **returns a new function** that can be used to do the scaling!

```javascript
//create the scale function
var diagramScale = d3.scaleLinear()
                     .domain([20,60]) //specify the domain
                     .range([0,120]); //specify the range

var result = diagramScale(50); //60, as above
var source = diagramScale.invert(60); //50, get the domain value from the range
```

Important: the returned `diagramScale` is a function! *Functions are values*, and in the `linearScale()` function returns a function as its result (instead of a string or an array).

You "set" the domain and range for the resulting function by calling the `domain()` and `range()` functions on it respectively. These functions take in an array of two or more values which are used to specify "stops" where a particular domain value will map to a particular range value. Any values in between these stops will be linearly interpolated (hence the "linear scale").

Note that it is also possible to use *colors* as range values, producing a nice gradient:

```
var scaleColor = d3.scaleLinear()
                  .domain([-100, 0, 100])
                  .range(['red', 'white', 'green']);  //using named colors

scaleColor(100);  //rgb(0, 128, 0), or green
scaleColor(0);    //rgb(255, 255, 255), or white
scaleColor(-50);  //rgb(255, 128, 128), or pink (between red and white)
```

It is also possible to specify more options for a scale function by calling additional methods on it. For example, `clamp()` will make sure a domain value doesn't go outside the range, and `nice()` will make sure the domain starts on nice round values. You can also use the `d3.min()` and `d3.max()` helper methods to perform a **reducing** operating on an array to get its minimum or maximum value (similar to the Python functions). This is useful when specify the domain values to be dependent on some external data.

D3 supports creating non-linear scaling functions as well. For example, `d3.scaleLog()` will produce a logarithmic mapping, and `d3.saleOrdinal()` will produce an ordinal mapping. See the documentation for a complete list of options.

19.4.1   Axes

Scales allow you to effectively position elements based on the data they represent. Additionally, you often would like to display the scales to the user, so that they know what values are associated with what positions. You do this by including **axes** (plural of **axis**) in your visualization. An axis is a ***visual representation of a scale*** that the user is able to see, and are used to explain to the *human* what the shapes in the visualization are depicting.

Since axes involve numerous elements to display (the axis bar, tick marks, labels on those tick marks, etc), D3 includes *helper functions* that can be used to easily create these elements: `d3.axisBottom()` creates an axis with the tick marks and labels on the bottom, `d3.axisLeft()` creates an axis with the ticks and labels on the left, etc. You pass these functions the *scale* that you want to create an axis for:

```
var xScale = d3.scaleLinear().domain([20,60]).range([0,120]);  //the scale
var xAxis = d3.axisBottom(xScale);  //make an axis for that scale
```

Like scales, an axis (returned by the `axisBottom()` function) **is itself a function**—you can think of it as an "axis creator" function. When called, the function will create all of the appropriate DOM elements (in the correct positions!) for the axis, *appending* those elements to the argument of the function:

```
var axisGroup = svg.append('g')  //create and position a group to hold the axis
    .attr('transform', 'translate('+axisXPosition+','+axisYPosition+')');
xAxis(axisGroup);  //create the axis in that group
```

However, since this requires creating a separate variable for the "parent" of the axis, it is more common to have the *parent element itself* execute the axis creator function **as a callback function** by using the `call()` method:

```
svg.append('g')  //create and position a group to hold the axis
    .attr('transform', 'translate('+axisXPosition+','+axisYPosition+')')
    .call(xAxis);  //call the axis creator function
```

With the `call()` method, the "executing" object (e.g., what you call the method on) will be passed in as the first argument to the callback (similar to a **pipe**)—and since that is what the axis generator function expected, it will be able to create the axis in the correct location!

- The `call()` method can also be used to "abstract" attribute modifications into separate functions for readability and ease of use:

  ```
  //a function that applies "styling" attributes to the selection
  function styleShape(selection){
  selection
      .attr('fill', 'gold')
      .attr('stroke', 'rebeccapurple')
      .attr('stroke-width', 5);
  }

  dataJoin.enter().append('rect')  //create new rects for each data element
      .call(styleShape) //apply styling to those elements!
  ```

As with *transitions* and *scales*, there are numerous methods you can use to customize the axis created by the axis function. The most common options involve styling the tick marks by calling the `ticks()` method on the axis. This method usually takes two arguments: the number of tick marks to include, and a format string specifying the format of the labels:

```
var xAxis = d3.axisBottom(xScale);  //make an axis for that scale
xAxis.ticks(7, '.0f')  //7 tick marks, format with 0 numbers after the decimal
```

The second argument is a formatting string similar to that used in Python string formatting. You can also specify this formatting using the `d3.format()` function. See this tool to experiment with options.

Resources

Note that some of these resources may not be up-to-date with D3 version 4.0 (what we are importing by default).

- D3 Tutorials (Official),
  - Three Little Circles (Bostock)
  - Thinking with Joins (Bostock)
  - General Update Pattern (Bostock)
  - Margin Convention
- Interactive Data Visualization for the Web (Murray)
  - Scales
  - Axes
- D3 Scales and Colors (v3, but a good summary of features)
- Interactive Information Visualization (Freeman). See in particular Chapter 12-14.

Appendix A

Markdown

Markdown syntax provides a simple way to describe the desired formatting of text documents. In fact, this book was written using Markdown! With only a small handful of options, Markdown allows you to format to your text (like making text **bold**, or *italics*), as well as provide structure to a document (such as headers or bullet-points). There are a number of programs and services that support the *rendering* of Markdown, including GitHub, Slack, and StackOverflow (though note the syntax may vary slightly across programs). In this chapter, you'll learn the basics of Markdown syntax, and how to leverage it to produce readable code documents.

A.1 Writing Markdown

Markdown is a lightweight markup language that is used to format and structure text. It is a kind of "code" that you write in order to *annotate* plain text: it lets the computer know that "this text is bold", "this text is a heading", etc. Compared to other markup languages, Markdown is easy to write and easy to read without getting in the way of the text itself. And because it's so simple to include, it's often used for formatting in web forums and services (like Wikipedia or StackOverflow). As a programmer, you'll use Markdown to create documentation and other supplementary materials that help explain your projects.

A.1.1 Text Formatting

At its most basic, Markdown is used to declare text formatting options. You do this by adding special symbols (punctuation) *around* the text you wish to "mark". For example, if you want text to be rendered as *italiccs*, you would

surround that text with underscores (`_`): you would type `_italics_`, and a
program would know to render that text as *italics*. You can see how this looks
in the below example (code on the left, rendered version on the right):
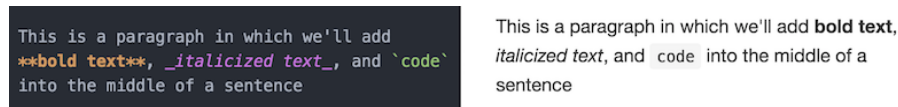

Figure A.1: Markdown text formatting.

There are a few different ways you can format text:

Syntax	Formatting
`_text_`	*italicized* using underscores (`_`)
`**text**`	**bolded** using two asterisks (`*`)
`` `text` ``	inline `code` with backticks (`` ` ``)
`~~text~~`	~~strike-through~~ using tildes (`~`)

A.1.2   Text Blocks

But Markdown isn't just about adding **bold** and *italics* in the middle of text—it
also enables you to create distinct blocks of formatted content (such as a header
or a chunk of code). You do this by adding a single symbol in front of the text.
Consider the below example:

As you can see, the document (right) is produced using the following Markdown
shorthand:

Syntax	Formatting
`#`	Header (use ## for 2nd-level, ### for 3rd, etc.)
` ``` `	Code section (3 back ticks) that encapsulate the code
`-`	Bulleted/unordered lists (hyphens)
`>`	Block quote

And as you might have guessed from this document, Markdown can even make
tables, create hyperlinks, and include images!

For more thorough lists of Markdown options, see the resources linked below.

Note that Slack will allow you to use Markdown as well, though it has slightly
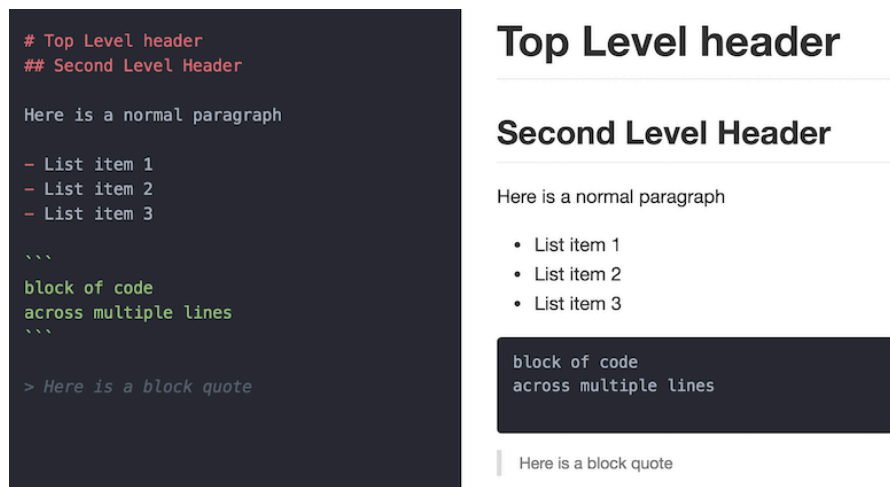different syntax. Luckily, the client gives you hints about what it supports:

Figure A.2: Markdown block formatting.

Figure A.3: Markdown in Slack.

A.2   Rendering Markdown

In order to view the *rendered* version of your Markdown-formatted syntax, you need to use a program that converts from Markdown into a formatted document. Luckily, GitHub will automatically render your Markdown files (which end with the `.md` extension), and Slack or StackOverflow will automatically format your messages.

However, it can be helpful to preview your rendered Markdown before posting code. The best way to do this is to write your marked code in a text-editor that supports preview rendering, such as **VS Code**.

- To preview what your rendered content will look like, simply open a Markdown file (`.md`) in VS Code. Then use the command palette to open the **Markdown Preview** (either in a new tab or in split-screen). Once this preview is open, it will automatically update to reflect any changes to the text!

Other options for rendering Markdown include:

- Many editors (such as Atom) also include automatic Markdown rendering, or have extensions to provide that functionality. Atom makes it easy to *Toggle Github Style* and make sure the rendered preview looks the same as it will when uploaded to GitHub.

- Stand-alone programs such as Macdown (Mac only) will also do the same work, often providing nicer looking editor windows.

- There are a variety of online Markdown editors that you can use for practice or quick tests. Dillinger is one of the nicer ones, but there are plenty of others if you're looking for something more specific.

- There are also a number of Google Chrome Extensions that will render Markdown files for you. For example, Markdown Reader, provides a simple rendering of a Markdown file (note it may differ slightly from the way GitHub would render the document). Once you've installed the Extension, you can drag-and-drop a `.md` file into a blank Chrome tab to view the formatted document. Double-click to view the raw code.

Resources

- Original Markdown Source
- GitHub Markdown Basics
- Slack Markdown
- StackOverflow Markdown