
COE 332: Software Engineering & Design

Texas Advanced Computing Center

May 13, 2022

COURSE SCHEDULE:

1	Unit 1: Onboarding and Essential Skills	3
2	Unit 2: Working with Common Data Formats	55
3	Unit 3: Best Practices in Python	71
4	Unit 4: Containerization and Automation	89
5	Unit 5: Introduction to APIs and Flask	115
6	Unit 6: Intro to Databases and Persistence, Containerizing Redis	143
7	Unit 7: Container Orchestration	157
8	Unit 8: Asynchronous Programming	187
9	Unit 9: Integration Testing, Continuous Integration	219
10	Unit 10: Special Topics	237
11	Homework 01	247
12	Homework 02	249
13	Homework 03	253
14	Homework 04	257
15	Midterm Project	261
16	Homework 05	265
17	Homework 06	267
18	Homework 07	269
19	Homework 08	271
20	Final Project	273
21	Additional Resources	277

The objective of this course is to introduce students to advanced computing concepts in software engineering, software systems design, cloud computing, distributed systems, and computational engineering. Through a series of assignments spanning the course of the semester, students will build a cloud-based, computational system to interact with a time series dataset and provide a web-accessible interface to their system.

UNIT 1: ONBOARDING AND ESSENTIAL SKILLS

We will use the first couple weeks of class to set expectations and describe requirements for the upcoming semester. We will begin to set up some necessary accounts and onboard students to the TACC / class ecosystem. We will also quickly review essential Linux, Python, and Version Control (Git) skills that will be necessary for the rest of this course.

1.1 Class Introduction

Welcome to:

COE 332 – Software Engineering & Design, Spring 2022
Department of Aerospace Engineering and Engineering Mechanics
The University of Texas at Austin

Instructors:

- Joe Allen, wallen@tacc.utexas.edu
- Joe Stubbs, jstubbs@tacc.utexas.edu
- Charlie Dey, charlie@tacc.utexas.edu

Time: Tues/Thur 11:00am - 12:30pm

Location: Virtual / Zoom (first two weeks of class), ASE 1.112 A (in person start date TBD)

Teaching Assistant: Geonyeong Lee, geon@utexas.edu

Important Links:

- Canvas: <https://utexas.instructure.com/courses/1326929>
- Class Repo: <https://coe-332-sp22.readthedocs.io/>
- Slack: <https://tacc-learn.slack.com/>

Catalog Description:

Covers methods and tools for planning, designing, implementing, validating and maintaining large software systems. May include project work to build a software system as a team, using appropriate software engineering tools and techniques.

Prerequisites:

Computational Engineering 322 with a grade of at least C-.

Knowledge, Skills, and Abilities Students Should Have Before Entering This Course:

This course assumes familiarity with the Python programming language and strong working knowledge of basic, high-level language programming concepts including data structures, conditionals, loops, and functions. We also assume a basic, working knowledge of the Linux command line. We will briefly review programming concepts in Linux and Python during the first week of class, the first homework assignment will be based on these topics, and we will make every effort to help students who are less familiar with these concepts. Ultimately, each student is expected to and responsible for mastering this material. This is not an introductory programming class and we will not have time to give a comprehensive treatment of all of these topics.

Knowledge, Skills, and Abilities Students Gain from this Course (Learning Outcomes):

The objective of this course is to introduce students to advanced computing concepts in software engineering, software systems design, cloud computing, distributed systems, and computational engineering. Through a series of assignments spanning the course of the semester, students will build a cloud-based, computational system to interact with a time series data set and provide a web-accessible interface to their system.

Relationship of Course to Program Outcomes:

This course contributes to the ABET Criterion 3 student outcomes that took effect with the Fall 2019 semester. For more information, see Criteria for Accrediting Engineering Programs, 2020-2021 at <https://www.abet.org/accreditation/accreditation-criteria/criteria-for-accrediting-engineering-programs-2020-2021/>

STUDENT OUTCOME	
1. an ability to identify, formulate, and solve complex engineering problems by applying principles of engineering, science, and mathematics	
2. an ability to apply engineering design to produce solutions that meet specified needs with consideration of public health, safety, and welfare, as well as global, cultural, social, environmental, and economic factors	
3. an ability to communicate effectively with a range of audiences	
4. an ability to recognize ethical and professional responsibilities in engineering situations and make informed judgments, which must consider the impact of engineering solutions in global, economic, environmental, and societal contexts	
5. an ability to function effectively on a team whose members together provide leadership, create a collaborative and inclusive environment, establish goals, plan tasks, and meet objectives	
6. an ability to develop and conduct appropriate experimentation, analyze and interpret data, and use engineering judgment to draw conclusions	
7. an ability to acquire and apply new knowledge as needed, using appropriate learning strategies	

ABET Program Criteria Achieved:

Program criteria are unique to each degree program and are to be compiled from the program criteria given for each degree program and listed in table format below. The faculty should check which of the program criteria are achieved in the course.

Criterion		Criterion		Criterion	
A. Aerodynamics		G. Orbital Mechanics		M. Preliminary/Conceptual Design	
B. Aerospace Materials		H. Space Environment		N. Other Design Content	
C. Structures		I. Attitude Determination and Control		O. Professionalism	
D. Propulsion		J. Telecommunications		P. Computer Usage	
E. Flight Mechanics		K. Space Structures			
F. Stability and Control		L. Rocket Propulsion			

Topics (subject to change):

- Software engineering and design practices in Python (d=30) (outcomes=1,2,3,4,5,6,7; criteria=M,N,O,P)
- Version control with Git (d=1)(outcomes=2,3,5; criteria=N,O,P)
- Unit testing, integration testing, continuous integration (d=4) (outcomes=1,2,6; criteria=M,N,O,P)
- Linux containers with Docker, container orchestration (d=4) (outcomes=1,2,7; criteria=M,N,O,P)
- HTTP and RESTful web services (d=5) (outcomes=1,2,6, criteria=M,N,P)
- Application programming interfaces (d=2) (outcomes=1,2,3; criteria=M,N,O,P)
- Virtualization with Kubernetes (d=4) (outcomes=1,2,7; criteria=M,N,P)
- Databases and Queues (d=4) (outcomes=1,2,6; criteria=M,N,P)
- Asynchronous programming patterns in distributed systems (d=2) (outcomes=1,2,5,6,7; criteria=M,N,P)
- Other advanced topics as time permits

Please note: In the course of learning and working through the above topics, students will be exposed to data sets from a variety of sources (e.g. <https://data.nasa.gov/browse>) representing other select Criterion (A-L) relevant to the ABET program accreditation. The exact data sets and types of data will depend on individual student interests.

Professionalism Topics:

Throughout the course, students will be taught to communicate professionally in the documentation of their software and in their software engineering & design projects. Effective communication in software projects is necessary to inform other engineers about the purpose or function of the project, and how to use it. Students will form teams to work together on the Final Projects (see description below) and will be required to address ethical and professional responsibilities in the course of working on their project.

Design Assignments (Final Project Description):

The Final Project will be a culmination of all materials covered in the class. Students will build a cloud-based, computational system to interact with a time series data set and provide a web-accessible interface to their system. Prior to working on the Final Project, students will form teams to work through a “design” phase where they must identify a primary data set to work on, describe API endpoints, diagram architecture components, and draft other major documentation components. In addition, students will be required to write in their own words what they think their ethical and professional responsibilities are as an engineer and how that relates to the Final Project. The project design will

be pitched to the instructors and given feedback / subject to approval. The Final Project will be due at the end of the semester in the form of a written report (e.g. pdf) and a GitHub repository containing the software and all support files. The full Final Project description will be posted on the class webpage.

Computer:

The entire course will be computer based. The instructors will provide a remote server for students to work on. Students are expected to have access to a personal / lab computer with a web browser and a terminal (or SCP client).

Text:

No textbook will be used for this course.

Class Format:

The class will be delivered in a hybrid format, utilizing both online and in-person experiences. Following UT's guidance, the first two weeks will be exclusively delivered via Zoom. During those two weeks we will continue to monitor for updated guidance from UT with the plan to eventually transition to in person if safe and appropriate. If / when we transition to in person lectures, we will continue to also offer the lectures via Zoom for those who do not wish to attend in person. This is subject to change.

Most class meetings will be comprised of lectures/demonstrations and hands-on labs. Students are expected to attend every lecture and actively participate in the hands-on labs during the class. The hands-on portions will often solve parts of homework assignments. Lecture materials with worked examples will be posted to the class website right before the class meeting. Additionally, there will be a class Slack channel for discussing ideas about the course with your fellow students.

Class Schedule (approximate, subject to change):

- Week 1: Onboarding, Linux, Python Review
- Week 2: Version Control, Working with JSON, CSV, XML
- Week 3: Unit Testing, Logging,
- Week 4: Intro to Containers, YAML
- Week 5: Advanced Containers, Docker Compose
- Week 6: HTTP, REST, Intro to Flask
- Week 7: Advanced Flask, Containerized Flask
- Week 8: Databases, Persistence in REST, **Midterm Project Due**
- Week of March 14 – Spring Break
- Week 9: Virtualization: Container Orchestration and Kubernetes
- Week 10: Virtualization: Container Orchestration and Kubernetes, cont.
- Week 11: Asynchronous Programming
- Week 12: Queues
- Week 13: Continuous Integration, Integration Testing
- Week 14: Special Topics
- Week 15: Special Topics - Final Week of Class
- Final Exam Day / Time: Thursday, May 12, 9:00 am-12:00 noon, **Final Project Due**

Grading:

Grades for the course will be based on the following:

- 30% Homework – Approximately 8-10 coding / software design assignments to be submitted via GitHub.

- 30% Midterm – A midterm design project will include concepts from the first half of the semester and build on the first 4-5 homework assignments. A written component will also be required.
- 40% Final Project - Students will form groups to work on a final class project consisting of a distributed, web-accessible, cloud system to interact with a time series data set. The project will draw from and build upon work done throughout the semester in homework assignments. The project will need to be pitched to the instructors for approval, and a written component will also be required.

Attendance:

Regular attendance is expected but absences will not count against the student's grades. We expect students to give us a week notice in advance of their absence if known ahead of time.

Office Hours:

Office hours will be for 1 hour immediately following the class and/or by appointment. We plan to use Slack for general communications and to help with the materials. <https://tacc-learn.slack.com/>

Important Dates:

Please refer to UT's academic calendar for important dates: <https://registrar.utexas.edu/calendars/21-22>

Special Notes:

The University of Texas at Austin provides upon request appropriate academic adjustments for qualified students with disabilities. For more information, contact the Office of the Dean of Students at 471-6259, 471-4641 TDD or the Cockrell School of Engineering Director of Students with Disabilities at 471-4321.

Evaluation:

Note that the Measurement and Evaluation Center forms for the Cockrell School of Engineering will be used during the last week of class to evaluate the course and the instructor. They will be conducted in an electronic format for Spring 2021. You may also want to note any other methods of evaluation you plan to employ.

Classroom Safety and COVID-19:

To help preserve our in person learning environment, the university recommends the following.

- Adhere to university mask guidance.
- Vaccinations are widely available, free and not billed to health insurance. The vaccine will help protect against the transmission of the virus to others and reduce serious symptoms in those who are vaccinated.
- Proactive Community Testing remains an important part of the university's efforts to protect our community. Tests are fast and free.
- Visit <https://protect.utexas.edu/> for more information.

Class Recordings:

Class recordings are reserved only for students in this class for educational purposes and are protected under FERPA. The recordings should not be shared outside the class in any form. Violation of this restriction by a student could lead to Student Misconduct proceedings. Guidance on public access to class recordings can be found [here](#).

1.1.1 Additional Help

Our main goal for this class is your success. Please contact us if you need help:

- Joe Allen, wallen@tacc.utexas.edu
- Joe Stubbs, jstubbs@tacc.utexas.edu
- Charlie Dey, charlie@tacc.utexas.edu

Important links:

- Canvas: <https://utexas.instructure.com/courses/1326929>
- Class Repo: <https://coe-332-sp22.readthedocs.io/>
- Slack: <https://tacc-learn.slack.com/>

1.2 Onboarding to TACC

The Texas Advanced Computing Center (TACC) at UT Austin designs and operates some of the world's most powerful computing resources. The center's mission is to enable discoveries that advance science and society through the application of advanced computing technologies.

We will be using cloud resources at TACC as our development environment. We will access the cloud resources via our SSH clients and TACC account credentials. **You will need a TACC username, password, and multifactor token for this class.**

Attention: Everyone please apply for a TACC account now using [this link](#). If you already have a TACC account, you can just use that. Send your TACC username to the course instructors via Slack or e-mail as soon as possible (see below).

To: {wallen, jstubbs, charlie} [at] tacc [dot] utexas [dot] edu
From: you
Subject: COE 332 TACC Account
Body: Please include your name, EID, TACC user name

1.2.1 About TACC

TACC is a Research Center, part of UT Austin, and located at the JJ Pickle Research Campus.

TACC at a Glance

Other TACC Activities

- Portals and gateways
- Web service APIs
- Rich software stacks
- Consulting
- Curation and analysis
- Code optimization
- Training and outreach

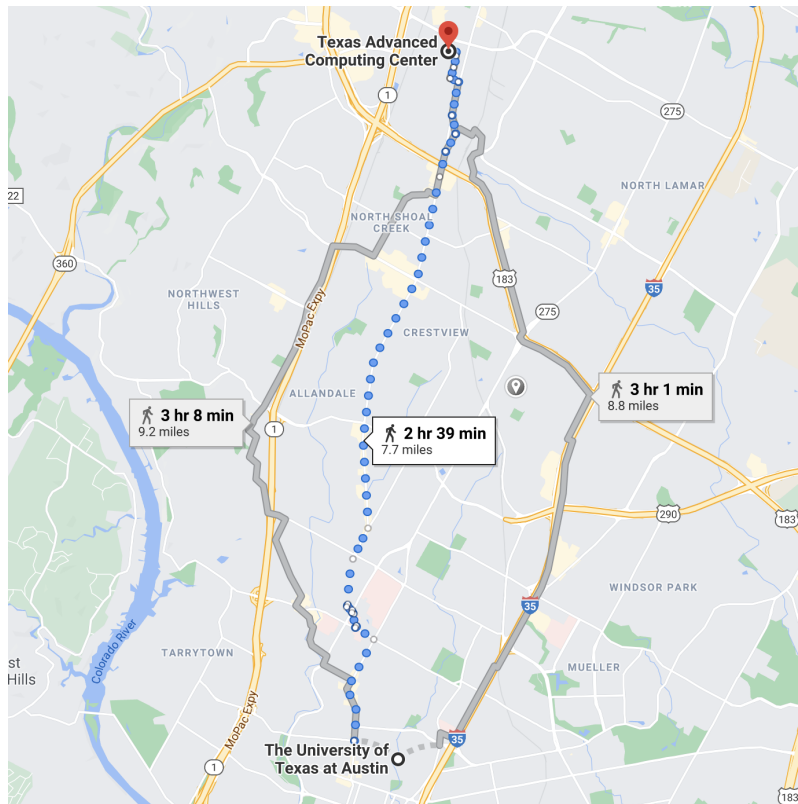


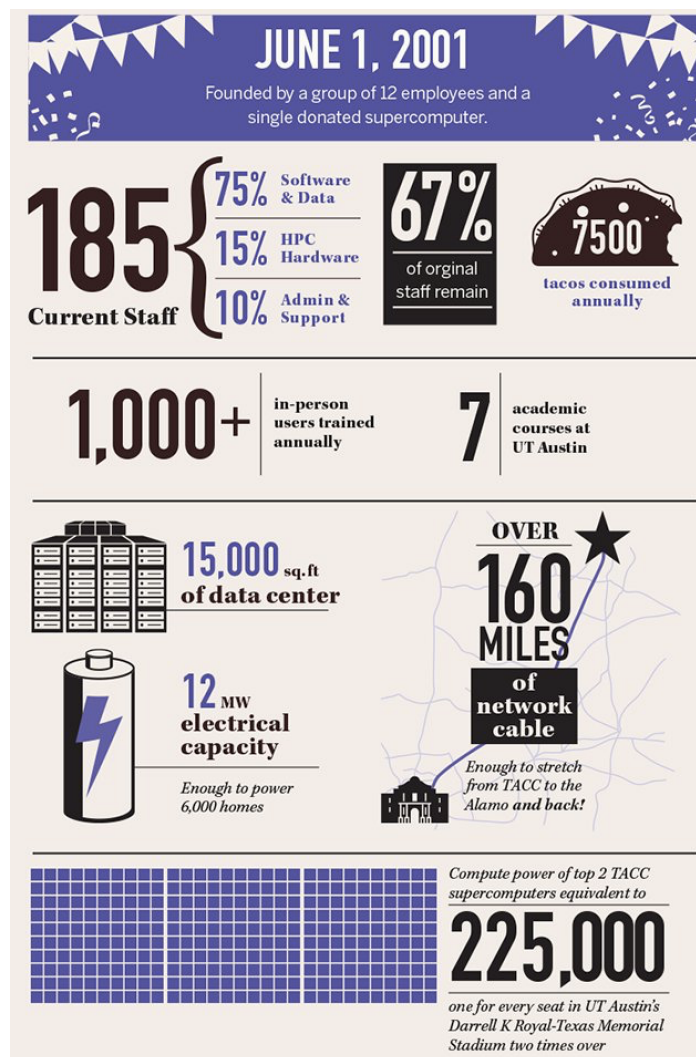
Fig. 1: A short 7.7 mile walk from main campus!

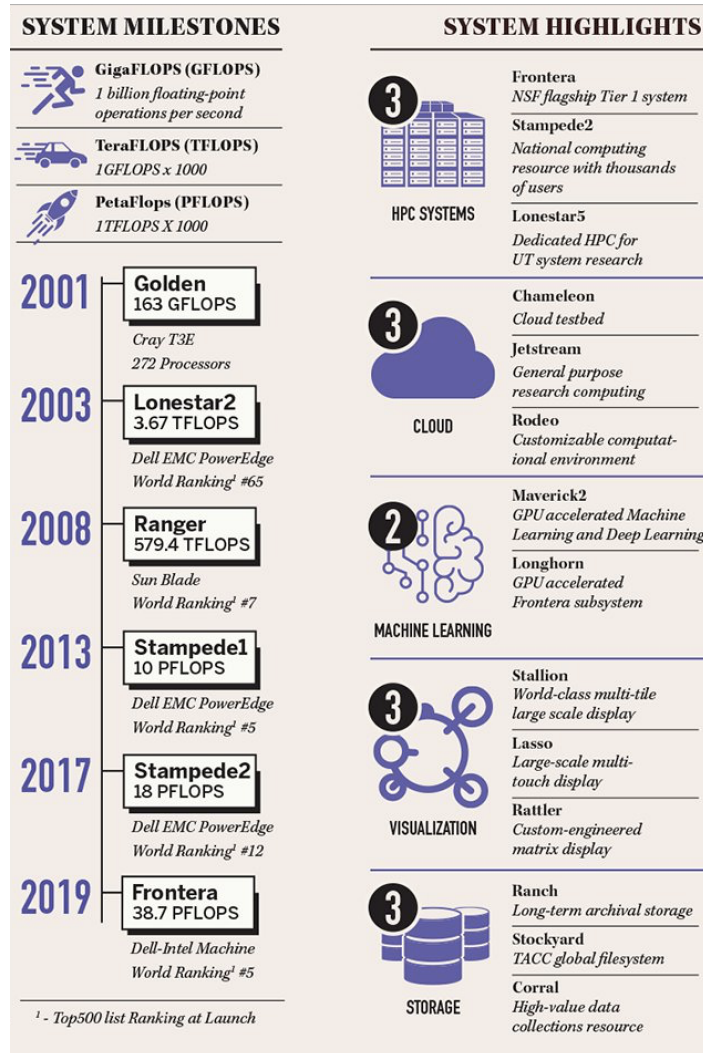


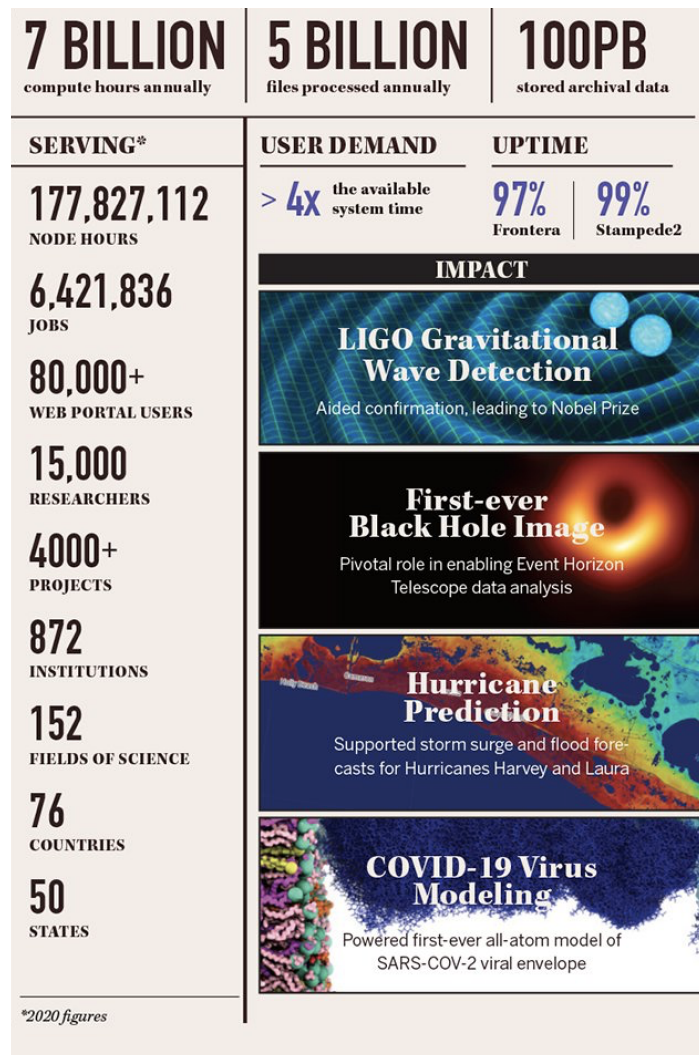
Fig. 2: One of two TACC buildings located at JJ Pickle.



Fig. 3: A tall person standing among taller Frontera racks.







- => [Learn more](#)

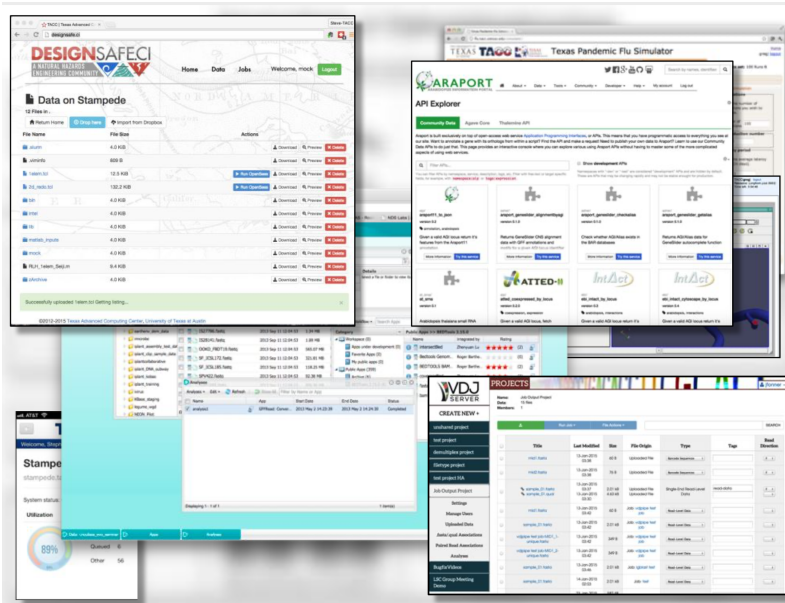


Fig. 4: Snapshot of a few of TACC's portal projects.

TACC Partnerships

- NSF: Leadership Class Computing Facility (LCCF)
- NSF: Extreme Science and Engineering Discovery Environment (XSEDE)
- UT Research Cyberinfrastructure (UTRC)
- TX Lonestar Education and Research Network (LEARN)
- Industry, [STAR Program](#)
- International, The International Collaboratory for Emerging Technologies
- => [Learn more](#)

Attention: Did you already e-mail your TACC username to the course instructors?

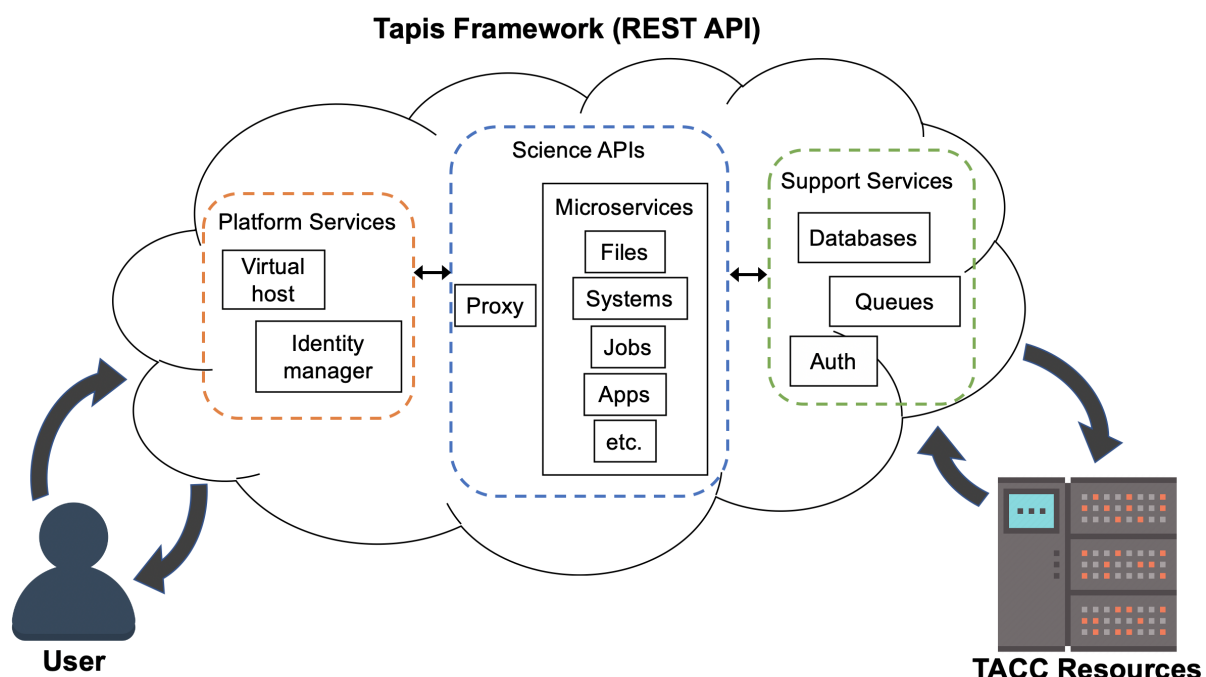
Which brings us to the question: **Why are we here teaching this class?**

1.2.2 Engineering Complex Systems in the Cloud

The Tapis Framework, developed at TACC, is a great example of a complex assembly of code with many moving parts, engineered to help researchers interact with high performance computing systems in streamlined and automated ways. Tapis empowers its users to:

- Authenticate using TACC (or other) credentials
- Manage, move, share, and publish data sets
- Run scientific code in batch jobs on clusters
- Set up event-driven processes
- *Many other things!*

The above description of Tapis and the below schematic diagram are both intentionally left a little bit vague as we will cover more of the specifics of Tapis later on in the semester.



Tip: Astute observers may notice that most, if not all, tools, technologies, and concepts that form the Tapis ecosystem show up somewhere in the agenda for COE 332.

So what can you do with Tapis?

Why would I want to build something similar?

Why should I learn how to use all of these tools and technologies?

Without concrete examples, it can seem rather esoteric. The vignette below hopefully illustrate how a carefully designed framework can be employed to tackle a real-world problem.

Real-Time Quantitative MRI

Problem: Quantitative analysis of MR images is typically performed after the patient has left the scanner. Corrupted or poor quality images can result in patient call backs, delaying disease intervention.

Importance: Real-time analytics of MRI scans can enable same-session quality control, reducing patient call backs, and it can enable precision medicine.

Approach: Faculty and staff from UTHealth - Houston and TACC used the Tapis framework to help develop an automated platform for real-time MRI.

Result: Scan data can now be automatically processed on high performance computing resources in real-time with no human intervention.

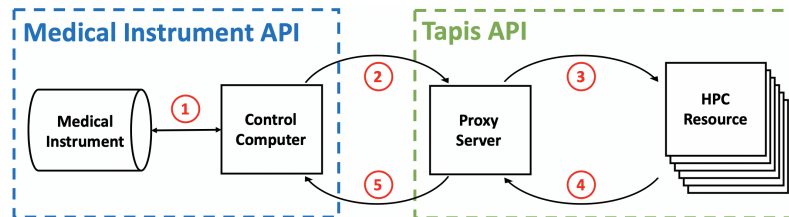


Fig. 5: Diagram of computer systems and APIs employed.

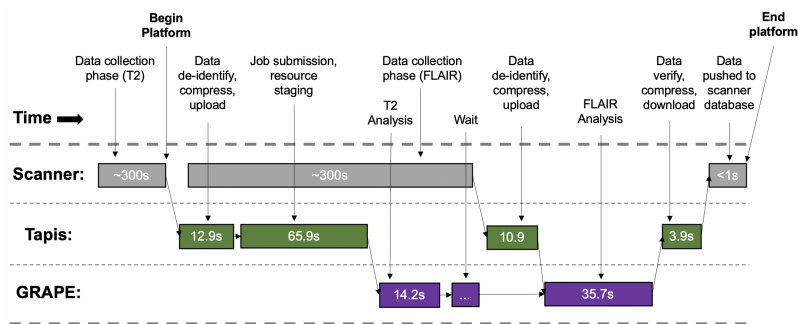


Fig. 6: Sample platform workflow for combining two images into one enhanced image.

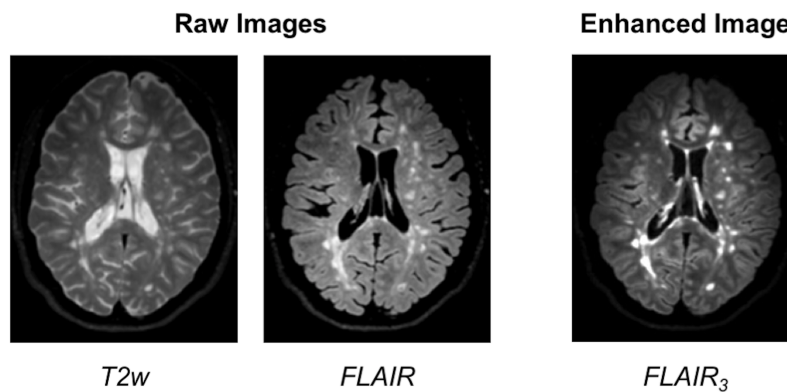


Fig. 7: Final image shows enhanced MS lesions.

Source: <https://dx.doi.org/10.1109/JBHI.2017.2771299>

Attention: If you already e-mailed your TACC account to the instructors, please go ahead and try the exercise below.

1.2.3 Engineering Complex Systems on ... Mars?

On April 19, 2021, the helicopter *Ingenuity* (part of NASA's Mars 2020 mission along with the rover *Perseverance*) completed the first ever “powered controlled extraterrestrial flight by an aircraft”. As of January 2022, it has made ~18 flights recording pictures, sound, position, and other data during flight.

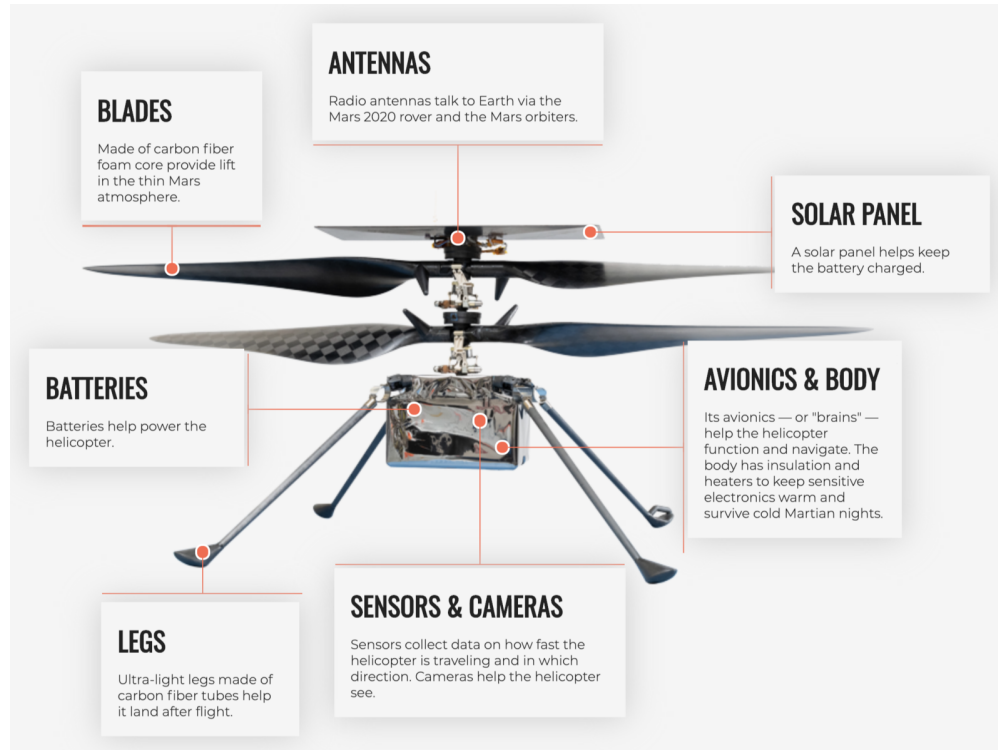


Fig. 8: Source: https://en.wikipedia.org/wiki/File:Anatomy_of_the_Mars_Helicopter.png

How do *Perseverance* and *Ingenuity* communicate to carry out missions and return that sensor data? The source code for the helicopter / rover themselves are not public (as far as I can tell), but NASA JPL credited a [long list](#) of open source code repositories on GitHub that the helicopter project depends on.

Included in the list are a lot of libraries and tools that we will be using this semester to build our cloud systems including: Linux, curl, pycurl, yaml, flask, click, pytest, jinja, requests, urllib3, werkzeug, (and many others). Looking at the packages, it seems pretty likely that the rover communicates with the helicopter through something similar to a REST API! We will all be building similar systems this semester.

Read more here: <https://github.com/readme/featured/nasa-ingenuity-helicopter>

1.2.4 Bringing it All Together

Hopefully these examples start to show you what kind of software projects we will be working on this semester. Each week will be introducing a new concept, tool, or technology that will slowly be building to a larger overall framework with many moving parts.

1.2.5 For Next Time

Using your SSH client, please try to log in to the class server **before the next class period**:

```
[local]$ ssh username@isp02.tacc.utexas.edu
Password:
TACC_Token:
Last login: Fri Jan 14 09:34:04 2022 from cpe-24-27-53-74.austin.res.rr.com
-----
Welcome to the Texas Advanced Computing Center
  at The University of Texas at Austin

** Unauthorized use/access is prohibited. **

If you log on to this computer system, you acknowledge your awareness
of and concurrence with the UT Austin Acceptable Use Policy. The
University will prosecute violators to the full extent of the law.

TACC Usage Policies:
http://www.tacc.utexas.edu/user-services/usage-policies/

TACC Support:
https://portal.tacc.utexas.edu/tacc-consulting

-----
Intel(R) Parallel Studio XE 2020 Update 1 for Linux*
Copyright (C) 2009-2020 Intel Corporation. All rights reserved.
[remote]$ hostname -f
isp02.tacc.utexas.edu      # success!
```

Note: In the above, replace 'username' with your TACC username.

1.3 Student Goals

When asked what their goals were for the course, students responded (*some goals were edited for typos and readability*):

Programming / Languages

- Hone my coding skills
- Build better code skills
- Become more comfortable with my programming skills
- Become more confident in my programming skills
- Gain better control of my programming skills and have the ability to implement them into advanced concepts by myself
- Improve my knowledge of coding enough for me to feel confident in my programming skills
- Be independent in my coding and become comfortable in my programming skills in a practical context
- Become more comfortable with Linux

- Enhance skills in Linux
- Better understanding of coding and more specifically python
- Get a deeper understanding of python
- Be comfortable in Python
- Get more proficient with python
- Learn Java and do practical assignments using it

Software Engineering / Design

- Learn more about software engineering
- Improve my software engineering skills
- Familiarize myself with software design
- Learn how to create professional, production worthy code
- Proper organization of large Python/Python-wrapped code systems
- Develop a solid understanding of the software techniques being taught in the class so that I can apply them not only on the assignments but on my own individual projects
- Grasp all of the concepts taught in this course well enough to both apply them in my own projects and appreciate their implementation in others' designs

Technologies

- Gain exposure to new skills and technology
- Learn new stuff and combined them with what I had then apply to the projects or any labs we have in the future
- Learn how to efficiently use git
- Understand and develop a REST API
- Understand containerization and become comfortable using Kubernetes
- Develop fundamental knowledge in cloud computing
- Obtain a strong foundation in the basics of cloud-based computing
- Learn more about distributed computing
- Flask apps, databases, asynchronous/queues, virtualization, integration, Docker, and REST API

Data Science

- Be able to create new ways to deal with complex data

Team Work

- Interact more with my peers and professors
- Work better on team assignments by organizing a reliable structure and schedule, and communicating with my teammates to complete everything in a timely fashion
- Learn how to work well with others on assignments and projects that resemble those I will do in the future in other classes or my career
- Work well on (interesting) team coding project
- Be comfortable enough with the concepts that if needed, I would be able to explain it well to others
- Take more of a leadership role in group projects

Career

- Pursuing internships in software engineering
- Apply my creativity to the technical skills I'm developing so that I can start my own projects
- Use the language in programs that are applicable to real-world things
- Broaden my knowledge in the programming world to thrive in internships/research I may have in the future
- Become proficient in python in order to attain an internship or complete a solo project
- Develop my skills and knowledge, which will get me closer to get into the technology field
- Be able to carry this information with me throughout college and early into my career
- Learn to grow my software development skills to be a successful software engineer
- Enjoy what I am learning

Can't Argue These Ones

- Achieve the best grade I can in this class
- Earn an A in the course

Good news: You are all in the right place.

1.4 Linux Essentials

To be successful in this class, students should be able to:

- Describe basic functions of essential Linux commands
- Use Linux commands to navigate a file system and manipulate files
- Transfer data to / from a remote Linux file system
- Edit files directly on a Linux system using a command line utility (e.g. vim, nano, emacs)

Topics covered in this module include:

- Creating and navigating folders (pwd, ls, mkdir, cd, rmdir)
- Creating and manipulating files (touch, rm, mv, cp)
- Looking at the contents of files (cat, more, less, head, tail, grep)
- Network and file transfers (hostname, whoami, logout, ssh, scp, rsync)
- Text editing with vim (insert mode, normal mode, navigating, saving, quitting)

1.4.1 Log in to the Class Server

To log in to `isp02.tacc.utexas.edu`, follow the instructions for your operating system or ssh client below.

Note: Replace username with your TACC username.

Mac / Linux

```
Open the application 'Terminal'  
ssh username@isp02.tacc.utexas.edu  
(enter password)  
(enter token)
```

Windows (use WSL or an SSH client like PuTTY)

```
Open the application 'PuTTY'  
enter Host Name: isp02.tacc.utexas.edu  
(click 'Open')  
(enter username)  
(enter password)  
(enter token)
```

If you can't access the class server yet, a local or web-based Linux environment will work for this guide. However, you will need to access the class server for future lectures.

Try this [Linux environment in a browser](#).

1.4.2 Creating and Navigating Folders

On a Windows or Mac desktop, our present location determines what files and folders we can access. I can “see” my present location visually with the help of the graphic interface - I could be looking at my Desktop, or the contents of a folder, for example. In a Linux command-line interface, we lack the same visual queues to tell us what our location is. Instead, we use a command - `pwd` (print working directory) - to tell us our present location. Try executing this command in the terminal:

```
$ pwd  
/home/wallen
```

This home location on the Linux filesystem is unique for each user, and it is roughly analogous to `C:\Users\username` on Windows, or `/Users/username` on Mac.

To see what files and folders are available at this location, use the `ls` (list) command:

```
$ ls
```

I have no files or folders in my home directory yet, so I do not get a response. We can create some folders using the `mkdir` (make directory) command. The words ‘folder’ and ‘directory’ are interchangeable:

```
$ mkdir folder1  
$ mkdir folder2  
$ mkdir folder3
```

```
$ ls  
folder1 folder2 folder3
```

Now we have some folders to work with. To “open” a folder, navigate into that folder using the `cd` (change directory) command. This process is analogous to double-clicking a folder on Windows or Mac:

```
$ pwd  
/home/wallen/  
$ cd folder1
```

(continues on next page)

(continued from previous page)

```
$ pwd
/home/wallen/folder1
```

Now that we are inside `folder1`, make a few sub-folders:

```
$ mkdir subfolderA
$ mkdir subfolderB
$ mkdir subfolderC
$ ls
subfolderA subfolderB subfolderC
```

Use `cd` to Navigate into `subfolderA`, then use `ls` to list the contents. What do you expect to see?

```
$ cd subfolderA
$ pwd
/home/wallen/folder1/subfolderA
$ ls
```

There is nothing there because we have not made anything yet. Next, we will navigate back to the home directory. So far we have seen how to navigate “down” into folders, but how do we navigate back “up” to the parent folder? There are different ways to do it. For example, we could specify the complete path of where we want to go:

```
$ pwd
/home/wallen/folder1/subfolderA
$ cd /home/wallen/folder1
$ pwd
/home/wallen/folder1/
```

Or, we could use a shortcut, `..`, which refers to the **parent folder** - one level higher than the present location:

```
$ pwd
/home/wallen/folder1
$ cd ..
$ pwd
/home/wallen
```

We are back in our home directory. Finally, use the `rmdir` (remove directory) command to remove folders. This will not work on folders that have any contents (more on this later):

```
$ mkdir junkfolder
$ ls
folder1 folder2 folder3 junkfolder
$ rmdir junkfolder
$ ls
folder1 folder2 folder3
```

Before we move on, let’s remove the directories we have made, using `rm -r` to remove our parent folder `folder1` and its subfolders. The `-r` command line option recursively removes subfolders and files located “down” the parent directory. `-r` is required for non-empty folders.

```
$ rm -r folder1
$ ls
folder2 folder3
```

Which command should we use to remove `folder2` and `folder3`?

```
$ rmdir folder2
$ rmdir folder3
$ ls
```

1.4.3 Creating and Manipulating Files

We have seen how to navigate around the filesystem and perform operations with folders. But, what about files? Just like on Windows or Mac, we can easily create new files, copy files, rename files, and move files to different locations. First, we will navigate to the home directory and create a few new folders and files with the `mkdir` and `touch` commands:

```
$ cd      # cd on an empty line will automatically take you back to the home directory
$ pwd
/home/wallen
$ mkdir folder1
$ mkdir folder2
$ mkdir folder3
$ touch file_a
$ touch file_b
$ touch file_c
$ ls
file_a  file_b  file_c  folder1  folder2  folder3
```

These files we have created are all empty. Removing a file is done with the `rm` (remove) command. Please note that on Linux file systems, there is no “Recycle Bin”. Any file or folder removed is gone forever and often un-recoverable:

```
$ touch junkfile
$ rm junkfile
```

Moving files with the `mv` command and copying files with the `cp` command works similarly to how you would expect on a Windows or Mac machine. The context around the move or copy operation determines what the result will be. For example, we could move and/or copy files into folders:

```
$ mv file_a folder1/
$ mv file_b folder2/
$ cp file_c folder3/
```

Before listing the results with `ls`, try to guess what the result will be.

```
$ ls
file_c folder1  folder2  folder3
$ ls folder1
file_a
$ ls folder2
file_b
$ ls folder3
file_c
```

Two files have been moved into folders, and `file_c` has been copied - so there is still a copy of `file_c` in the home directory. Move and copy commands can also be used to change the name of a file:

```
$ cp file_c file_c_copy
$ mv file_c file_c_new_name
```

By now, you may have found that Linux is very unforgiving with typos. Generous use of the <Tab> key to auto-complete file and folder names, as well as the <UpArrow> to cycle back through command history, will greatly improve the experience. As a general rule, try not to use spaces or strange characters in files or folder names. Stick to:

```
A-Z    # capital letters
a-z    # lowercase letters
0-9    # digits
-      # hyphen
_      # underscore
.      # period
```

Before we move on, let's clean up once again by removing the files and folders we have created. Do you remember the command for removing non-empty folders?

```
$ rm -r folder1
$ rm -r folder2
$ rm -r folder3
```

How do we remove `file_c_copy` and `file_c_new_name`?

```
$ rm file_c_copy
$ rm file_c_new_name
```

1.4.4 Looking at the Contents of Files

Everything we have seen so far has been with empty files and folders. We will now start looking at some real data. Navigate to your home directory, then issue the following `cp` command to copy a public file on the server to your local space:

```
$ cd ~      # the tilde ~ is also a shortcut referring to your home directory
$ pwd
/home/wallen
$ cp /usr/share/dict/words .
$ ls
words
```

Try to use <Tab> to autocomplete the name of the file. Also, please notice the single dot `.` at the end of the copy command, which indicates that you want to cp the file to `.`, this present location (your home directory).

This `words` file is a standard file that can be found on most Linux operating systems. It contains 479,828 words, each word on its own line. To see the contents of a file, use the `cat` command to print it to screen:

```
$ cat words
1080
10-point
10th
11-point
12-point
16-point
18-point
1st
2
20-point
```

This is a long file! Printing everything to screen is much too fast and not very useful. We can use a few other commands to look at the contents of the file with **more** control:

```
$ more words
```

Press the <Enter> key to scroll through line-by-line, or the <Space> key to scroll through page-by-page. Press q to quit the view, or <Ctrl+c> to force a quit if things freeze up. A % indicator at the bottom of the screen shows your progress through the file. This is still a little bit messy and fills up the screen. The **less** command has the same effect, but is a little bit cleaner:

```
$ less words
```

Scrolling through the data is the same, but now we can also search the data. Press the / forward slash key, and type a word that you would like to search for. The screen will jump down to the first match of that word. The n key will cycle through other matches, if they exist.

Finally, you can view just the beginning or the end of a file with the **head** and **tail** commands. For example:

```
$ head words
$ tail words
```

The > and >> shortcuts in Linux indicate that you would like to redirect the output of one of the commands above. Instead of printing to screen, the output can be redirected into a file:

```
$ cat words > words_new.txt
$ head words > first_10_lines.txt
```

A single greater than sign > will redirect and **overwrite** any contents in the target file. A double greater than sign >> will redirect and **append** any output to the end of the target file.

One final useful way to look at the contents of files is with the **grep** command. **grep** searches a file for a specific pattern, and returns all lines that match the pattern. For example:

```
$ grep "banana" words
banana
bananaquit
bananas
cassabanana
```

Although it is not always necessary, it is safe to put the search term in quotes.

1.4.5 Network and File Transfers

In order to login or transfer files to a remote Linux file system, you must know the hostname (unique network identifier) and the username. If you are already on a Linux file system, those are easy to determine using the following commands:

```
$ whoami
wallen
$ hostname -f
isp02.tacc.utexas.edu
```

Given that information, a user would remotely login to this Linux machine using **ssh** in a Terminal:


```
[local]$ ssh wallen@isp02.tacc.utexas.edu
(enter password)
(enter token)
[isp02]$
```

Windows users would typically use the program **PuTTY** (or another SSH client) to perform this operation. Logging out of a remote system is done using the `logout` command, or the shortcut `<Ctrl+d>`:

```
[isp02]$ logout
[local]$
```

Copying files from your local computer to your home folder on ISP would require the `scp` command (Windows users use a client “WinSCP”):

```
[local]$ scp my_file wallen@isp02.tacc.utexas.edu:/home/wallen/
(enter password)
(enter token)
```

In this command, you specify the name of the file you want to transfer (`my_file`), the username (`wallen`), the hostname (`isp02.tacc.utexas.edu`), and the path you want to put the file (`/home/wallen/`). Take careful notice of the separators including spaces, the `@` symbol, and the `:`.

Copy files from ISP to your local computer using the following:

```
[local]$ scp wallen@isp02.tacc.utexas.edu:/home/wallen/my_file ./
(enter password)
(enter token)
```

Instead of files, full directories can be copied using the “recursive” flag (`scp -r ...`). The `rsync` tool is an advanced copy tool that is useful for synching data between two sites. Although we will not go into depth here, example `rsync` usage is as follows:

```
$ rsync -azv local remote
$ rsync -azv remote local
```

This is just the basics of copying files. See example [scp usage](#) and example [rsync usage](#) for more info.

1.4.6 Text Editing with VIM

VIM is a text editor used on Linux file systems.

Open a file (or create a new file if it does not exist):

```
$ vim file_name
```

There are two “modes” in VIM that we will talk about today. They are called “insert mode” and “normal mode”. In insert mode, the user is typing text into a file as seen through the terminal (think about typing text into TextEdit or Notepad). In normal mode, the user can perform other functions like save, quit, cut and paste, find and replace, etc. (think about clicking the menu options in TextEdit or Notepad). The two main keys to remember to toggle between the modes are `i` and `Esc`.

Entering VIM insert mode:

```
> i
```

Entering VIM normal mode:

`> Esc`

A summary of the most important keys to know for normal mode are:

Navigating the file:

arrow keys	move up, down, left, right
Ctrl+u	page up
Ctrl+d	page down
0	move to beginning of line
\$	move to end of line
gg	move to beginning of file
G	move to end of file
:N	move to line N

Saving and quitting:

:q	quit editing the file
:q!	quit editing the file without saving
:w	save the file, continue editing
:wq	save and quit

1.4.7 Review of Topics Covered

Part 1: Creating and navigating folders

Command	Effect
pwd	print working directory
ls	list files and directories
ls -l	list files in column format
mkdir dir_name/	make a new directory
cd dir_name/	navigate into a directory
rmdir dir_name/	remove an empty directory
rm -r dir_name/	remove a directory and its contents
. or ./	refers to the present location
.. or ../	refers to the parent directory

Part 2: Creating and manipulating files

Command	Effect
<code>touch file_name</code>	create a new file
<code>rm file_name</code>	remove a file
<code>rm -r dir_name/</code>	remove a directory and its contents
<code>mv file_name dir_name/</code>	move a file into a directory
<code>mv old_file new_file</code>	change the name of a file
<code>mv old_dir/ new_dir/</code>	change the name of a directory
<code>cp old_file new_file</code>	copy a file
<code>cp -r old_dir/ new_dir/</code>	copy a directory
<code><Tab></code>	autocomplete file or folder names
<code><UpArrow></code>	cycle through command history

Part 3: Looking at the contents of files

Command	Effect
<code>cat file_name</code>	print file contents to screen
<code>cat file_name >> new_file</code>	redirect output to new file
<code>more file_name</code>	scroll through file contents
<code>less file_name</code>	scroll through file contents
<code>head file_name</code>	output beginning of file
<code>tail file_name</code>	output end of a file
<code>grep pattern file_name</code>	search for 'pattern' in a file
<code>~/</code>	shortcut for home directory
<code><Ctrl+c></code>	force interrupt
<code>></code>	redirect and overwrite
<code>>></code>	redirect and append

Part 4: Network and file transfers

Command	Effect
<code>hostname -f</code>	print hostname
<code>whoami</code>	print username
<code>ssh username@hostname</code>	remote login
<code>logout</code>	logout
<code>scp local remote</code>	copy a file from local to remote
<code>scp remote local</code>	copy a file from remote to local
<code>rsync -azv local remote</code>	sync files between local and remote
<code>rsync -azv remote local</code>	sync files between remote and local
<code><Ctrl+d></code>	logout of host

Part 5: Text editing with VIM

Command	Effect
<code>vim file.txt</code>	open "file.txt" and edit with vim
<code>i</code>	toggle to insert mode
<code><Esc></code>	toggle to normal mode
<code><arrow keys></code>	navigate the file
<code>:q</code>	quit ending the file
<code>:q!</code>	quit editing the file without saving
<code>:w</code>	save the file, continue editing
<code>:wq</code>	save and quit

1.4.8 Additional Resources

- Practice Linux commands safely in a web-based emulator
- This is a good summary of the important commands you need to know
- Practice VIM in a web browser
- Practice VIM on the command line by typing `vimtutor`

1.5 Python Refresher

To be successful in this class, students should be able to:

- Write and execute Python code on the class server
- Use variables, lists, and dictionaries in Python
- Write conditionals using a variety of comparison operators
- Write useful while and for loops
- Arrange code into clean, well organized functions
- Read input from and write output to a file
- Import and use standard and non-standard Python libraries

Topics covered in this module include:

- Data types and variables (ints, floats, bools, strings, `type()`, `print()`)
- Arithmetic operations (+, -, *, /, **, %, //)
- Lists and dictionaries (creating, interpreting, appending)
- Conditionals and control loops (comparison operators, `if/elif/else`, `while`, `for`, `break`, `continue`, `pass`)
- Functions (defining, passing arguments, returning values)
- File handling (`open`, `with`, `read()`, `readline()`, `strip()`, `write()`)
- Importing libraries (`import`, `random`, `names`, `pip`)

1.5.1 Log in to the Class Server

To log in to `isp02.tacc.utexas.edu`, follow the instructions for your operating system or ssh client below.

Note: Replace `username` with your TACC username.

Mac / Linux

```
Open the application 'Terminal'
ssh username@isp02.tacc.utexas.edu
(enter password)
(enter token)
```

Windows (use WSL or an SCP client like PuTTY)

```
Open the application 'PuTTY'
enter Host Name: isp02.tacc.utexas.edu
(click 'Open')
(enter username)
(enter password)
(enter token)
```

If you can't access the class server yet, a local or web-based Python 3 environment will work for this guide. However, you will need to access the class server for future lectures.

Try this [Python 3 environment in a browser](#).

Note: For the first few sections below, we will be using the Python interpreter in *interactive mode* to try out different things. Later on when we get to more complex code, we will be saving the code in files (scripts) and invoking the interpreter non-interactively.

1.5.2 Data Types and Variables

Start up the interactive Python interpreter:

```
[isp02]$ python3
Python 3.6.8 (default, Aug 7 2019, 17:28:10)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-39)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Tip: To exit the interpreter, type `quit()`.

The most common data types in Python are similar to other programming languages. For this class, we probably only need to worry about **integers**, **floats**, **booleans**, and **strings**.

Assign some values to variables by doing the following:

```
>>> my_int = 5
>>> my_float = 5.0
>>> my_bool = True      # or False, notice capital letters
>>> my_string = 'Hello, world!'
```

In Python, you don't have to declare type. Python figures out the type automatically. Check using the `type()` function:

```
>>> type(my_int)
<class 'int'>
>>> type(my_float)
<class 'float'>
>>> type(my_bool)
<class 'bool'>
>>> type(my_string)
<class 'str'>
```

Print the values of each variable using the `print()` function:

```
>>> print(my_int)
5
>>> print('my_int')
my_int
```

(Try printing the others as well). And, notice what happens when we print with and without single quotes? What is the difference between `my_int` and `'my_int'`?

You can convert between types using a few different functions. For example, when you read in data from a file, numbers are often read as strings. Thus, you may want to convert the string to integer or float as appropriate:

```
>>> str(my_int)      # convert int to string
>>> str(my_float)    # convert float to string
>>> int(my_string)   # convert string to int
>>> float(my_string) # convert string to float
>>>
>>> value = 5
>>> print(value)
5
>>> type(value)
<class 'int'>
>>> new_value = str(value)
>>> print(new_value)
'5'
>>> type(new_value)
<class 'str'>
```

1.5.3 Arithmetic Operations

Next, we will look at some basic arithmetic. You are probably familiar with the standard operations from other languages:

Operator	Function	Example	Result
+	Addition	1+1	2
-	Subtraction	9-5	4
*	Multiplication	2*2	4
/	Division	8/4	2
**	Exponentiation	3**2	9
%	Modulus	5%2	1
//	Floor division	5//2	2

Try a few things to see how they work:

```
>>> print(2+2)
>>> print(355/113)
>>> print(10%9)
>>> print(3+5*2)
>>> print('hello' + 'world')
>>> print('some' + 1)
>>> print('number' * 5)
```

Also, carefully consider how arithmetic options may affect type:

```
>>> number1 = 5.0/2
>>> type(number1)
<class 'float'>
>>> print(number1)
2.5
>>> number2 = 5/2
>>> type(number2)
<class 'float'>
>>> print(number2)
2.5
>>> print(int(number2))
2
```

1.5.4 Lists and Dictionaries

Lists are a data structure in Python that can contain multiple elements. They are ordered, they can contain duplicate values, and they can be modified. Declare a list with square brackets as follows:

```
>>> my_shape_list = ['circle', 'heart', 'triangle', 'square']
>>> type(my_shape_list)
<class 'list'>
>>> print(my_shape_list)
['circle', 'heart', 'triangle', 'square']
```

Access individual list elements:

```
>>> print(my_shape_list[0])
circle
>>> type(my_shape_list[0])
<class 'str'>
>>> print(my_shape_list[2])
triangle
```

Create an empty list and add things to it:

```
>>> my_number_list = []
>>> my_number_list.append(5)      # 'append()' is a method of the list class
>>> my_number_list.append(6)
>>> my_number_list.append(2)
>>> my_number_list.append(2**2)
>>> print(my_number_list)
[5, 6, 2, 4]
>>> type(my_number_list)
<class 'list'>
>>> type(my_number_list[1])
<class 'int'>
```

Lists are not restricted to containing one data type. Combine the lists together to demonstrate:

```
>>> my_big_list = my_shape_list + my_number_list
>>> print(my_big_list)
['circle', 'heart', 'triangle', 'square', 5, 6, 2, 4]
```

Another way to access the contents of lists is by slicing. Slicing supports a start index, stop index, and step taking the form: `mylist[start:stop:step]`. Only the first colon is required. If you omit the start, stop, or `:step`, it is assumed you mean the beginning, end, and a step of 1, respectively. Here are some examples of slicing:

```
>>> mylist = ['thing1', 'thing2', 'thing3', 'thing4', 'thing5']
>>> print(mylist[0:2])      # returns the first two things
['thing1', 'thing2']
>>> print(mylist[:2])      # if you omit the start index, it assumes the beginning
['thing1', 'thing2']
>>> print(mylist[-2:])     # returns the last two things (omit the stop index and it_
↪assumes the end)
['thing4', 'thing5']
>>> print(mylist[:])      # returns the entire list
['thing1', 'thing2', 'thing3', 'thing4', 'thing5']
>>> print(mylist[:2])     # return every other thing (step = 2)
['thing1', 'thing3', 'thing5']
```

Note: If you slice from a list, it returns an object of type list. If you access a list element by its index, it returns an object of whatever type that element is. The choice of whether to slice from a list, or iterate over a list by index, will depend on what you want to do with the data.

Dictionaries are another data structure in Python that contain key:value pairs. They are always unordered, they cannot contain duplicate keys, and they can be modified. Create a new dictionary using curly brackets:

```
>>> my_shape_dict = {
...     'most_favorite': 'square',
...     'least_favorite': 'circle',
...     'pointiest': 'triangle',
...     'roundest': 'circle'
... }
>>> type(my_shape_dict)
<class 'dict'>
>>> print(my_shape_dict)
{'most_favorite': 'square', 'least_favorite': 'circle', 'pointiest': 'triangle',
↪'roundest': 'circle'}
>>> print(my_shape_dict['most_favorite'])
square
```

As your preferences change over time, so to can values stored in dictionaries:

```
>>> my_shape_dict['most_favorite'] = 'rectangle'
>>> print(my_shape_dict['most_favorite'])
rectangle
```

Add new key:value pairs to the dictionary as follows:

```
>>> my_shape_dict['funniest'] = 'squircle'
>>> print(my_shape_dict['funniest'])
squircle
```

Many other methods exist to access, manipulate, interpolate, copy, etc., lists and dictionaries. We will learn more about them out as we encounter them later in this course.

1.5.5 Conditionals and Control Loops

Python **comparison operators** allow you to add conditions into your code in the form of `if / elif / else` statements. Valid comparison operators include:

Operator	Comparison	Example	Result
<code>==</code>	Equal	<code>1==2</code>	False
<code>!=</code>	Not equal	<code>1!=2</code>	True
<code>></code>	Greater than	<code>1>2</code>	False
<code><</code>	Less than	<code>1<2</code>	True
<code>>=</code>	Greater than or equal to	<code>1>=2</code>	False
<code><=</code>	Less Than or equal to	<code>1<=2</code>	True

A valid conditional statement might look like:

```
>>> num1 = 10
>>> num2 = 20
>>>
>>> if (num1 > num2):           # notice the colon
...     print('num1 is larger') # notice the indent
... elif (num2 > num1):
...     print('num2 is larger')
... else:
...     print('num1 and num2 are equal')
```

In addition, conditional statements can be combined with **logical operators**. Valid logical operators include:

Operator	Description	Example
<code>and</code>	Returns True if both are True	<code>a < b and c < d</code>
<code>or</code>	Returns True if at least one is True	<code>a < b or c < d</code>
<code>not</code>	Negate the result	<code>not(a < b)</code>

For example, consider the following code:

```
>>> num1 = 10
>>> num2 = 20
>>>
>>> if (num1 < 100 and num2 < 100):
...     print('both are less than 100')
... else:
...     print('at least one of them is not less than 100')
```

While loops also execute according to conditionals. They will continue to execute as long as a condition is True. For example:

```
>>> i = 0
>>>
>>> while (i < 10):
...     print( f'i = {i}' )      # literal string interpolation
...     i = i + 1
```

The `break` statement can also be used to escape loops:

```
>>> i = 0
>>>
>>> while (i < 10):
...     print( f'i = {i}' )
...     i = i + 1
...     if (i==5):
...         break
...     else:
...         continue
```

For loops in Python are useful when you need to execute the same set of instructions over and over again. They are especially great for iterating over lists:

```
>>> my_shape_list = ['circle', 'heart', 'triangle', 'square']
>>>
>>> for shape in my_shape_list:
...     print(shape)
>>>
>>> for shape in my_shape_list:
...     if (shape == 'circle'):
...         pass                                # do nothing
...     else:
...         print(shape)
```

You can also use the `range()` function to iterate over a range of numbers:

```
>>> for x in range(10):
...     print(x)
>>>
>>> for x in range(10, 100, 5):
...     print(x)
>>>
>>> for a in range(3):
...     for b in range(3):
...         for c in range(3):
...             print( f'{a} + {b} + {c} = {a+b+c}' )
```

Note: The code is getting a little bit more complicated now. It will be better to stop running in the interpreter's interactive mode, and start writing our code in Python scripts.

1.5.6 Functions

Functions are blocks of codes that are run only when we call them. We can pass data into functions, and have functions return data to us. Functions are absolutely essential to keeping code clean and organized.

On the command line, use a text editor to start writing a Python script:

```
[isp02]$ vim function_test.py
```

Enter the following text into the script:

```
1 def hello_world():
2     print('Hello, world!')
3
4 hello_world()
```

After saving and quitting the file, execute the script (Python code is not compiled - just run the raw script with the python3 executable):

```
[isp02]$ python3 function_test.py
Hello, world!
```

Note: Future examples from this point on will assume familiarity with using the text editor and executing the script. We will just be showing the contents of the script and console output.

More advanced functions can take parameters and return results:

```
1 def add5(value):
2     return(value + 5)
3
4 final_number = add5(10)
5 print(final_number)
```

```
15
```

Pass multiple parameters to a function:

```
1 def add5_after_m multiplying(value1, value2):
2     return( (value1 * value2) + 5)
3
4 final_number = add5_after_m multiplying(10, 2)
5 print(final_number)
```

```
25
```

It is a good idea to put your list operations into a function in case you plan to iterate over multiple lists:

```
1 def print_ts(mylist):
2     for x in mylist:
3         if (x[0] == 't'):      # a string (x) can be interpreted as a list of chars!
4             print(x)
5
6 list1 = ['circle', 'heart', 'triangle', 'square']
7 list2 = ['one', 'two', 'three', 'four']
8
9 print_ts(list1)
10 print_ts(list2)
```

```
triangle
two
three
```

There are many more ways to call functions, including handing an arbitrary number of arguments, passing keyword / unordered arguments, assigning default values to arguments, and more.

1.5.7 File Handling

The `open()` function does all of the file handling in Python. It takes two arguments - the *filename* and the *mode*. The possible modes are read (`r`), write (`w`), append (`a`), or create (`x`).

For example, to read a file do the following:

```
1 with open('/usr/share/dict/words', 'r') as f:
2     for x in range(5):
3         print(f.readline())
```

```
1080
10-point
10th
11-point
12-point
```

Tip: By opening the file with the `with` statement above, you get built in exception handling, and it automatically will close the file handle for you. It is generally recommended as the best practice for file handling.

You may have noticed in the above that there seems to be an extra space between each word. What is actually happening is that the file being read has newline characters on the end of each line (`\n`). When read into the Python script, the original new line is being printed, followed by another newline added by the `print()` function. Stripping the newline character from the original string is the easiest way to solve this problem:

```
1 with open('/usr/share/dict/words', 'r') as f:
2     for x in range(5):
3         print(f.readline().strip('\n'))
```

```
1080
10-point
10th
11-point
12-point
```

Read the whole file and store it as a list:

```
1 words = []
2
3 with open('/usr/share/dict/words', 'r') as f:
4     words = f.read().splitlines()           # careful of memory usage
5
6 for x in range(5):
7     print(words[x])
```

```
1080
10-point
10th
11-point
12-point
```

Write output to a new file on the file system; make sure you are attempting to write somewhere where you have permissions to write:

```
1 my_shapes = ['circle', 'heart', 'triangle', 'square']
2
3 with open('my_shapes.txt', 'w') as f:
4     for shape in my_shapes:
5         f.write(shape)
```

```
(in my_shapes.txt)
circlehearttriangle-square
```

You may notice the output file is lacking in newlines this time. Try adding newline characters to your output:

```
1 my_shapes = ['circle', 'heart', 'triangle', 'square']
2
3 with open('my_shapes.txt', 'w') as f:
4     for shape in my_shapes:
5         f.write( f'{shape}\n' )
```

```
(in my_shapes.txt)
circle
heart
triangle
square
```

Now notice that the original line in the output file is gone - it has been overwritten. Be careful if you are using write (w) vs. append (a).

1.5.8 Importing Libraries

The Python built-in functions, some of which we have seen above, are useful but limited. Part of what makes Python so powerful is the huge number and variety of libraries that can be *imported*. For example, if you want to work with random numbers, you have to import the ‘random’ library into your code, which has a method for generating random numbers called ‘random’.

```
1 import random
2
3 for i in range(5):
4     print(random.random())
```

```
0.47115888799541383
0.5202615354150987
0.8892412583071456
0.7467080997595558
0.025668541754695906
```

More information about using the `random` library can be found in the [Python docs](#)

Some libraries that you might want to use are not included in the official Python distribution - called the *Python Standard Library*. Libraries written by the user community can often be found on [PyPI.org](#) and downloaded to your local environment using a tool called `pip3`.

For example, if you wanted to download the `names` library and use it in your Python code, you would do the following:

```
[isp02]$ pip3 install --user names
Collecting names
  Downloading https://files.pythonhosted.org/packages/44/4e/
  ↪f9cb7ef2df0250f4ba3334fbdabaa94f9c88097089763d8e85ada8092f84/names-0.3.0.tar.gz (789kB)
    100% || 798kB 1.1MB/s
Installing collected packages: names
  Running setup.py install for names ... done
Successfully installed names-0.3.0
```

Notice the library is installed above with the `--user` flag. The class server is a shared system and non-privileged users can not download or install packages in root locations. The `--user` flag instructs `pip3` to install the library in your own home directory.

```
1 import names
2
3 for i in range(5):
4     print(names.get_full_name())
```

```
Johnny Campbell
Lawrence Webb
Johnathan Holmes
Mary Wang
Jonathan Henry
```

1.5.9 Exercises

Test your understanding of the materials above by attempting the following exercises.

- Create a list of ~10 different integers. Write a function (using modulus and conditionals) to determine if each integer is even or odd. Print to screen each digit followed by the word 'even' or 'odd' as appropriate.
- Using nested for loops and if statements, write a program that iterates over every integer from 3 to 100 (inclusive) and prints out the number only if it is a prime number.
- Create three lists containing 10 integers each. The first list should contain all the integers sequentially from 1 to 10 (inclusive). The second list should contain the squares of each element in the first list. The third list should contain the cubes of each element in the first list. Print all three lists side-by-side in three columns. E.g. the first row should contain 1, 1, 1 and the second row should contain 2, 4, 8.
- Write a script to read in `/usr/share/dict/words` and print just the last 10 lines of the file. Write another script to only print words beginning with the letters "pyt".

1.5.10 Additional Resources

- [The Python Standard Library](#)
- [PEP 8 Python Style Guide](#)
- [Python3 environment in a browser](#)
- [Jupyter Notebooks in a browser](#)

1.6 Version Control with Git: Part 1

In the next two modules, we will look at the version control system **Git**. Of the numerous version control systems available (Git, Subversion, CVS, Mercurial), Git seems to be the most popular, and we generally find that it is great for:

- Collaborating with others on code
- Supporting multiple concurrent versions (branches)
- Tagging releases or snapshots in time
- Restoring previous versions of files
- What it lacks in user-friendliness it makes up for in good documentation
- Intuitive web platforms available

After going through the two Git modules, students should be able to:

- Create a new Git repository hosted on GitHub
- Clone a repository, commit and push changes to the repository
- Track the history of changes in files in a Git repository
- Demonstrate a basic understanding of forking, branching, tags, and pull requests
- Work collaboratively with others on the content in a Git repository

GitHub is a web platform where you can host and share Git repositories (“repos”). Repositories can be public or private. Much of what we will do with this section requires you to have a GitHub account.

1.6.1 The Basics of Git

Version control systems start with a base version of the document and then record changes you make each step of the way. You can think of it as a recording of your progress: you can rewind to start at the base document and play back each change you made, eventually arriving at your more recent version.

Fig. 9: Changes are saved sequentially.

Once you think of changes as separate from the document itself, you can then think about “playing back” different sets of changes on the base document, ultimately resulting in different versions of that document. For example, two users can make independent sets of changes on the same document.

Fig. 10: Different versions can be saved.

Unless there are conflicts, you can even incorporate two sets of changes into the same base document.

Fig. 11: Multiple versions can be merged.

A version control system is a tool that keeps track of these changes for us, effectively creating different versions of our files. It allows us to decide which changes will be made to the next version (each record of these changes is called a “commit”, and keeps useful metadata about them. The complete history of commits for a particular project and their metadata make up a “repository”. Repositories can be kept in sync across different computers, facilitating collaboration among different people.

1.6.2 Setting up Git

Log on to the class ISP server and check which version of Git is in your PATH.

Note: Below, replace `username` with your TACC username.

```
[local]$ ssh username@isp02.tacc.utexas.edu  # use your account
(enter password)
(enter token)

[isp02]$ which git
/usr/bin/git
$ git --version
git version 1.8.3.1
```

When we use Git on a new computer for the first time, we need to configure a few things. Below are a few examples of configurations we will set as we get started with Git:

- Our name and email address,
- And that we want to use these settings globally (i.e. for every project).

On a command line, Git commands are written as `git verb`, where `verb` is what we actually want to do. Here is how we set up our environment:

```
[isp02]$ git config --global user.name "Joe Allen"
[isp02]$ git config --global user.email "wallen@tacc.utexas.edu"
```

Please use your own name and email address associated with your GitHub account. This user name and email will be connected with your subsequent Git activity, which means that any changes pushed to [GitHub](#), [Bitbucket](#), [GitLab](#) or another Git host server in the future will include this information.

Tip: A key benefit of Git is that it is platform agnostic. You can use it equally to interact with the same files from your laptop, from a lab computer, or from a cluster.

1.6.3 Create a New Repository on the Command Line

First, let's navigate to our home directory and create a folder for this class and for working with Git (if you haven't done it already):

```
[isp02]$ cd ~/
[isp02]$ mkdir coe-332      # you may already have a folder for this class
[isp02]$ cd coe-332
[isp02]$ mkdir my-first-git-repo
[isp02]$ cd my-first-git-repo/
[isp02]$ pwd
/home/wallen/coe-332/my-first-git-repo
```

Then we will use a Git command to initialize this directory as a new Git repository - or a place where Git can start to organize versions of our files.

```
[isp02]$ git init
Initialized empty Git repository in /home/wallen/coe-332/my-first-git-repo/.git/
```

If we use `ls -a`, we can see that Git has created a hidden directory called `.git`:

```
[isp02]$ ls -a
./  ../  .git/
```

Use the `find` command to get a overview of the contents of the `.git/` directory:

```
[isp02]$ find .git/
.git
.git/refs
.git/refs/heads
.git/refs/tags
.git/branches
.git/description
.git/hooks
.git/hooks/applypatch-msg.sample
.git/hooks/commit-msg.sample
.git/hooks/post-update.sample
.git/hooks/pre-applypatch.sample
.git/hooks/pre-commit.sample
.git/hooks/pre-push.sample
.git/hooks/pre-rebase.sample
.git/hooks/prepare-commit-msg.sample
.git/hooks/update.sample
.git/info
.git/info/exclude
.git/HEAD
.git/config
.git/objects
.git/objects/pack
.git/objects/info
```

Git uses this special sub-directory to store all the information about the project, including all files and sub-directories located within the project's directory. If we ever delete the `.git` sub-directory, we will lose the project's history. We can check that everything is set up correctly by asking Git to tell us the status of our project:

```
[isp02]$ git status
# On branch main
#
# Initial commit
nothing to commit (create/copy files and use "git add" to track)
```

As you see, there is “nothing to commit” because there are no files in here to track. To make things more interesting, let’s copy in a few of the Python scripts we were working on (or any other files) and check the status again:

```
[isp02]$ cp ~/coe-332/python-test/*.py ./
[isp02]$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       python_test_1.py
#       python_test_2.py
nothing added to commit but untracked files present (use "git add" to track)
```

Note: If you are using a different version of `git`, the exact wording of the output might be slightly different.

EXERCISE

- Explore the files and folders in the `.git/` directory
- Can you find any files with plain text info / meta data?

1.6.4 Tracking Changes

We will use this repository track some changes we are about to make to our example Python scripts. Above, Git mentioned that it found several “Untracked files”. This means there are files in this current directory that Git isn’t keeping track of. We can instruct Git to start tracking a file using `git add`:

```
[isp02]$ git add python_test_1.py
[isp02]$ git status
# On branch main
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   python_test_1.py
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
```

(continues on next page)

(continued from previous page)

```
#
# python_test_2.py
```

1.6.5 Commit Changes to the Repo

Git now knows that it's supposed to keep track of `python_test_1.py`, but it hasn't recorded these changes as a commit yet. To get it to do that, we need to run one more command:

```
[isp02]$ git commit -m "started tracking first Python script"
[main (root-commit) 344ec9f] started tracking first Python script
1 file changed, 29 insertions(+)
create mode 100644 python_test_1.py
```

When we run `git commit`, Git takes everything we have told it to save by using `git add` and stores a copy permanently inside the special `.git` directory. This permanent copy is called a “commit” (or “revision”) and its short identifier is `344ec9f`. Your commit may have another identifier.

We use the `-m` flag (“m” for “message”) to record a short, descriptive, and specific comment that will help us remember later on what we did and why. Good commit messages start with a brief (<50 characters) statement about the changes made in the commit. Generally, the message should complete the sentence “If applied, this commit will” *<commit message here>*. If you want to go into more detail, add a blank line between the summary line and your additional notes. Use this additional space to explain why you made changes and/or what their impact will be.

If we run `git status` now:

```
[isp02]$ git status
# On branch main
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       python_test_2.py
nothing added to commit but untracked files present (use "git add" to track)
```

We find one remaining untracked files.

EXERCISE

Do a `git add <file>` followed by a `git commit -m "descriptive message"` for the untracked file. Also do a `git status` in between each command.

1.6.6 Check the Project History

If we want to know what we've done recently, we can ask Git to show us the project's history using `git log`:

```
[isp02]$ git log
commit 3d5d6e2c6d23aa4fb3b800b535db6a228759866e
Author: Joe Allen <wallen@tacc.utexas.edu>
Date:   Wed Jan 27 23:06:03 2021 -0600

    added python_test_2.py
```

(continues on next page)

(continued from previous page)

```
commit 344ec9fde550c6e009697b07298919946ff991f9
Author: Joe Allen <wallen@tacc.utexas.edu>
Date:   Wed Jan 27 23:00:17 2021 -0600

    started tracking first Python script
```

The command `git log` lists all commits made to a repository in reverse chronological order. The listing for each commit includes:

- the commit's full identifier (which starts with the same characters as the short identifier printed by the `git commit` command earlier),
- the commit's author,
- when it was created,
- and the log message Git was given when the commit was created.

1.6.7 Making Further Changes

Now suppose we make a change to one of the files we are tracking. Edit the `python_test_1.py` script your favorite text editor and add some random comments into the script:

```
[isp02]$ vim python_test_1.py
# make some changes in the script
# save and quit
```

When we run `git status` now, it tells us that a file it already knows about has been modified:

```
[isp02]$ git status
# On branch main
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   python_test_1.py
#
no changes added to commit (use "git add" and/or "git commit -a")
```

The last line is the key phrase: “no changes added to commit”. We have changed this file, but we haven’t told Git we will want to save those changes (which we do with `git add`) nor have we saved them (which we do with `git commit`). So let’s do that now. It is good practice to always review our changes before saving them. We do this using `git diff`. This shows us the differences between the current state of the file and the most recently saved version:

```
[isp02]$ git diff python_test_1.py
diff --git a/python_test_1.py b/python_test_1.py
index 5d986e9..21877cb 100644
--- a/python_test_1.py
+++ b/python_test_1.py
@@ -18,7 +18,7 @@ def check_char_match(str1, str2):
     else:
         return( f'{str1} match FAILS' )
```

(continues on next page)

(continued from previous page)

```

+## random comments inserted here
with open('states.json', 'r') as f:
    states = json.load(f)

```

The output is cryptic because it is actually a series of commands for tools like editors and `patch` telling them how to reconstruct one file given the other. If we break it down into pieces:

- The first line tells us that Git is producing output similar to the Unix `diff` command comparing the old and new versions of the file.
- The second line tells exactly which versions of the file Git is comparing: `5d986e9` and `21877cb` are unique computer-generated labels for those versions.
- The third and fourth lines once again show the name of the file being changed.
- The remaining lines are the most interesting, they show us the actual differences and the lines on which they occur. In particular, the `+` marker in the first column shows where we added lines.

After reviewing our change, it's time to commit it:

```

[isp02]$ git add python_test_1.py
[isp02]$ git commit -m "added a descriptive comment"
[main 8d5f563] added a descriptive comment
1 file changed, 1 insertion(+), 1 deletion(-)
[isp02]$ git status
# On branch main
nothing to commit, working directory clean

```

Git insists that we add files to the set we want to commit before actually committing anything. This allows us to commit our changes in stages and capture changes in logical portions rather than only large batches. For example, suppose we're adding a few citations to relevant research to our thesis. We might want to commit those additions, and the corresponding bibliography entries, but *not* commit some of our work drafting the conclusion (which we haven't finished yet).

1.6.8 Directories in Git

There are a couple important facts you should know about directories in Git. First, Git does not track directories on their own, only files within them. Try it for yourself:

```

[isp02]$ mkdir directory
[isp02]$ git status
[isp02]$ git add directory
[isp02]$ git status

```

Note, our newly created empty directory `directory` does not appear in the list of untracked files even if we explicitly add it (via `git add`) to our repository.

Second, if you create a directory in your Git repository and populate it with files, you can add all files in the directory at once by:

```

[isp02]$ git add <directory-with-files>

```

Tip: A trick for tracking an empty directory with Git is to add a hidden file to the directory. People sometimes will label this `.gitcanary`. Adding and committing that file to the repo's history will cause the directory it is in to also be

tracked.

1.6.9 Restoring Old Versions of Files

We can save changes to files and see what we've changed — now how can we restore older versions of things? Let's suppose we accidentally overwrite our file:

```
[isp02]$ echo "" > python_test_1.py
[isp02]$ cat python_test_1.py
```

Now `git status` tells us that the file has been changed, but those changes haven't been staged:

```
[isp02]$ git status
# On branch main
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   python_test_1.py
#
no changes added to commit (use "git add" and/or "git commit -a")
```

We can put things back the way they were by using `git checkout` and referring to the *most recent commit* of the working directory by using the identifier `HEAD`:

```
[isp02]$ git checkout HEAD python_test_1.py
[isp02]$ cat python_test_1.py
import random
...etc
```

As you might guess from its name, `git checkout` checks out (i.e., restores) an old version of a file. In this case, we're telling Git that we want to recover the version of the file recorded in `HEAD`, which is the last saved commit. If we want to go back even further, we can use a commit identifier instead:

```
[isp02]$ git log
commit 8d5f563fa20060f4f4be2e10ec5cbc3c22fe92559
Author: Joe Allen <wallen@tacc.utexas.edu>
Date:   Wed Jan 27 23:15:46 2021 -0600

    added a descriptive comment

commit 3d5d6e2c6d23aa4fb3b800b535db6a228759866e
Author: Joe Allen <wallen@tacc.utexas.edu>
Date:   Wed Jan 27 23:06:03 2021 -0600

    adding python_test_2.py

commit 344ec9fde550c6e009697b07298919946ff991f9
Author: Joe Allen <wallen@tacc.utexas.edu>
Date:   Wed Jan 27 23:00:17 2021 -0600

    started tracking first Python script
```

```
[isp02]$ git checkout 344ec9f python_test_1.py
# now you have a copy of python_test_1.py without that comment we added
```

Again, we can put things back the way they were by using `git checkout`:

```
[isp02]$ git checkout HEAD python_test_1.py
# back to the most recent version
```

1.6.10 Summing Up

To summarize the first Git module, the commands we covered were:

```
git config      # Get and set repository or global options
git init        # Create an empty Git repository or reinitialize an existing one
git status      # Show the working tree status
git add         # Add file contents to the index
git commit      # Record changes to the repository
git diff        # Show changes between commits, commit and working tree, etc
git log         # Show commit logs
git checkout    # Checkout a branch or paths to the working tree
```

The key takeaway is the general workflow of *making some changes* => `git add` => `git commit`. If you think of Git as taking snapshots of changes over the life of a project, `git add` specifies what will go in a snapshot (putting things in the staging area), and `git commit` actually takes the snapshot, and makes a permanent record of it (as a commit).

1.6.11 Additional Resources

- Some of the materials in this module were based on [Software Carpentry DOI: 10.5281/zenodo.57467](#).
- [GitHub Glossary](#)
- [About Branches](#)
- [About Pull Requests](#)
- [About Licenses](#)
- [GitFlow Model](#)
- [More on different git workflows](#)

1.7 Version Control with Git: Part 2

In the first Git module, we learned to work independently with Git repositories on the local command line. In this second part, we will focus on using the GitHub web interface and collaborating with others.

1.7.1 Link a Local Repository to GitHub

Version control really comes into its own when we begin to collaborate with other people. We already have most of the machinery we need to do this; the only thing missing is to copy changes from one repository to another.

Systems like Git allow us to move work between any two repositories. In practice, though, it's easiest to use one copy as a central hub, and to keep it on the web rather than on someone's laptop. Most programmers use hosting services like GitHub, Bitbucket, or GitLab to hold those main copies.

Let's start by sharing the changes we've made to our current project with the world. Log in to GitHub, then click on the icon in the top right corner to create a new repository:

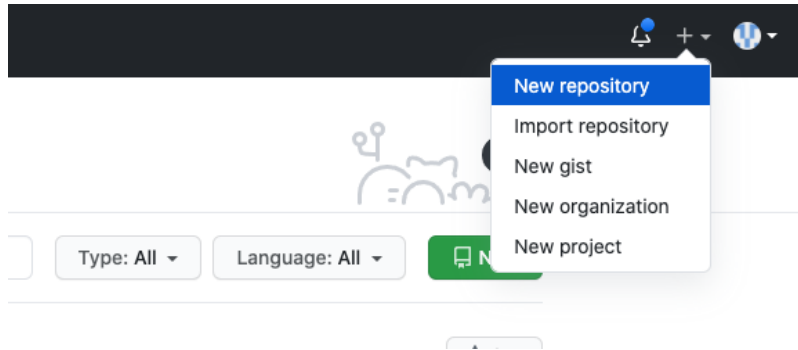


Fig. 12: Click 'New repository'.

As soon as the repository is created, GitHub displays a page with a URL and some information on how to configure your local repository. Provide a name for your new repository like `python-test` (or whatever you want, it doesn't have to match the name of your local folder).

Note that our local repository still contains our earlier work on `python_test_1.py` and other files, but the remote repository on GitHub doesn't contain any memory of `python_test_1.py` yet. The next step is to connect and sync the two repositories. We do this by making the GitHub repository a "remote" for the local repository. The home page of the repository on GitHub includes the string we need to identify it:

...or push an existing repository from the command line

```
git remote add origin git@github.com:wjallen/python-test.git
git branch -M main
git push -u origin main
```

Fig. 13: Follow the instructions for pushing an existing repository.

Back on ISP in the local Git repo, link it to the repo on GitHub and confirm the link was created:

```
[isp02]$ git remote add origin git@github.com:wjallen/python-test.git
[isp02]$ git remote -v
origin  git@github.com:wjallen/python-test.git (fetch)
origin  git@github.com:wjallen/python-test.git (push)
```

Attention: Make sure to use the URL for your repository instead of the one listed here.

The name `origin` is a local nickname for your remote repository. We could use something else if we wanted to, but `origin` is by far the most common choice.

Another key step is to set up SSH keys for authentication. GitHub no longer allows simple username / password authentication from the command line. To set up SSH keys, click on:

Your account => Settings => SSH and GPG keys => New SSH key

In the “Title” box, add a memorable name for this key like “isp02”. In the “Key” box, cut and paste the contents of your existing public key on the class server. You can find it by executing the command:

```
[isp02]$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQDF8crdmqgk2GhRmsLPcREWjzc9zb2B....
.....
```

Once the SSH key is set up, this command will push the changes from our local repository to the repository on GitHub:

```
[isp02]$ git branch -M main
[isp02]$ git push -u origin main
Warning: Permanently added the ECDSA host key for IP address '140.82.112.4' to the list
of known hosts.
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 223 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@github.com:wjallen/python-test.git
 * [new branch]      main -> main
Branch main set up to track remote branch main from origin.
```

Now that the repositories are synced, your development workflow has evolved to include the `git push` operation. From here on, if you make changes to your code, you can expect to follow the changes with the commands:

```
# Make some edits to "example_file.py"
[isp02]$ git status
[isp02]$ git add example_file.py
[isp02]$ git commit -m "description of changes"
[isp02]$ git push
```

1.7.2 Clone the Repository

Spend a few minutes browsing the web interface for GitHub. Now, anyone can make a full copy of `my_first_repo` including all the commit history by performing:

```
[isp02]$ git clone git@github.com:wjallen/python-test.git
Cloning into 'python-test'...
remote: Enumerating objects: 15, done.
remote: Counting objects: 100% (15/15), done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 15 (delta 4), reused 15 (delta 4), pack-reused 0
Unpacking objects: 100% (15/15), done.
```

If the repository on GitHub gets ahead of your local repository, i.e. it has some changes in it that someone else pushed from somewhere else, or you pushed from a different machine, then you can try to update your local repository to pull the changes back down.

```
[isp02]$ git remote update    # checks to see if there are updates in the remote
[isp02]$ git pull             # pulls those updates down to local
```

Warning: If you have changes in local files that conflict with the remote repository (i.e. the repository on GitHub), the `git pull` will fail and you have found your way into a “merge conflict”. *Good luck!*

1.7.3 Git / Version Control Concepts

Let’s take a quick intermission to lean some important definitions (most of these things can easily be managed in the GitHub web interface):

Fork

A fork is a personal copy of another user’s repository that lives on your account. Forks allow you to freely make changes to a project without affecting the original. Forks remain attached to the original, allowing you to submit a pull request to the original’s author to update with your changes. You can also keep your fork up to date by pulling in updates from the original.

Branch

A branch is a parallel version of a repository. It is contained within the repository, but does not affect the primary or main branch allowing you to work freely without disrupting the “live” version. When you’ve made the changes you want to make, you can merge your branch back into the main branch to publish your changes. For more information, see [About branches](#).

Tag

Git has the ability to tag specific points in history as being important. Typically people use this functionality to mark release points (v1.0, and so on).

Pull Request / Merge Request

Pull requests are proposed changes to a repository submitted by a user and accepted or rejected by a repository’s collaborators. Like issues, pull requests each have their own discussion forum. For more information, see [About pull requests](#).

1.7.4 Collaborating with Others

A public platform like GitHub makes it easier than ever to collaborate with others on the content of a repository. You can have as many local copies of a repository as you want, but there is only one “origin” repository - the repository hosted on GitHub. Other repositories may fall behind the origin, or have changes that are ahead of the origin. A common model for juggling multiple repositories where separate individuals are working on different features is the [GitFlow model](#):

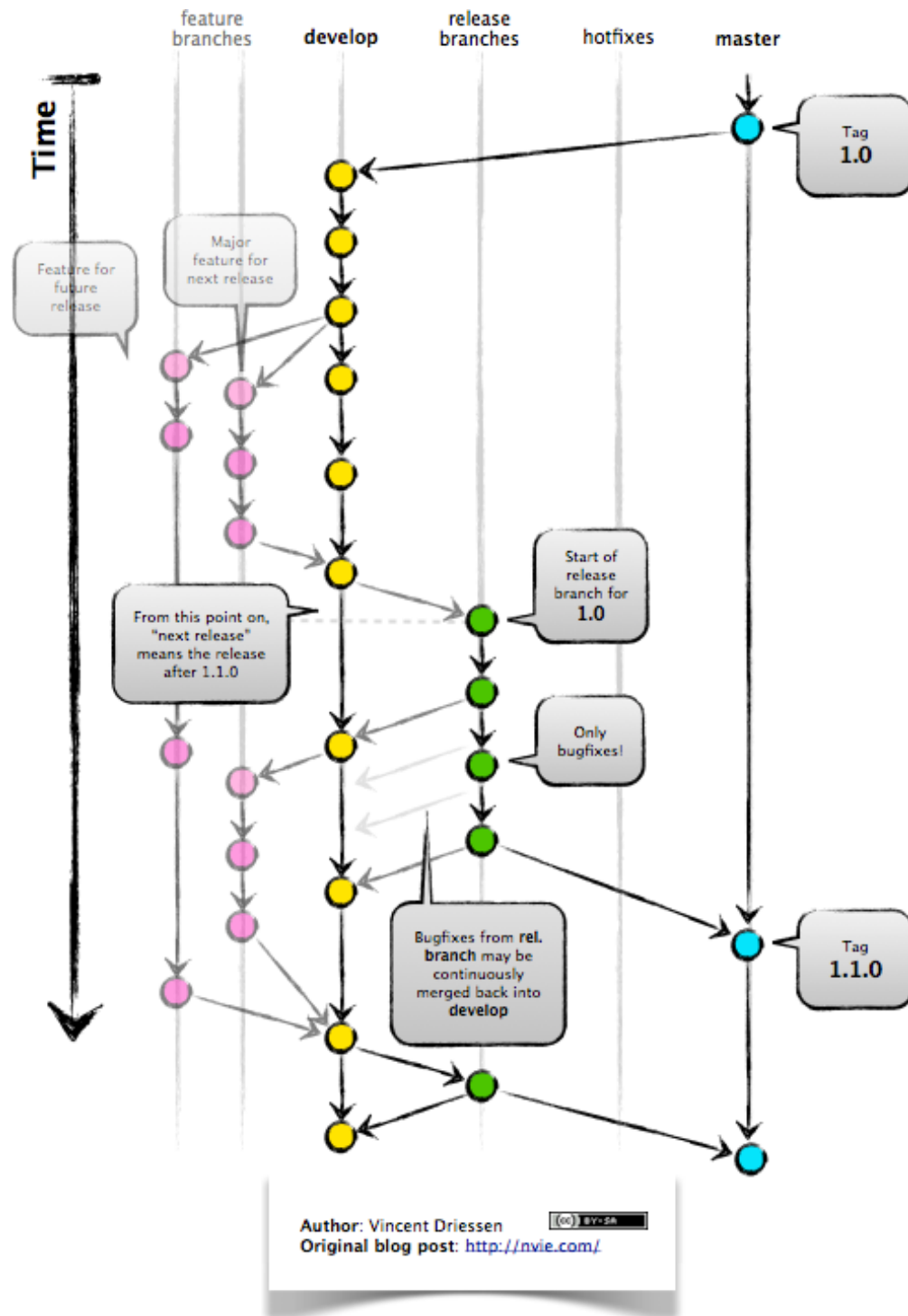


Fig. 14: GitFlow model

EXERCISE

Let's work on a **branch** plus **pull request** in the GitHub web interface.

- Locate and navigate to your repository
- Click on the branch / tag navigator near the top right and start typing in a new name to create a new branch
- By default, it should switch you to the new branch. Click on one of your files and edit it (e.g. add a comment) directly in the web interface
- Navigate to the "Pull requests" tab and click on "New pull request"
- Make sure to select the original branch "main" as the base, and your new branch as the downstream for comparison
- Review the code changes and make everything is as expected before choosing "Create pull request"
- Now the owner of the repository (you) can accept the pull request, merging the edits into the main branch

What are the differences between a "merge commit", "squash and merge", and "rebase and merge"? The differences are subtle. All will result in the edits getting merged into the main branch. It is mostly a stylistic thing, and the best method depends on whatever the rest of developers agree to use. More info on the differences [here](#).

EXERCISE

Let's next work on a **fork** plus **pull request** in the GitHub web interface.

- Navigate to this repository: <https://github.com/jagaither/coe-332-forking-demo>
- Click the "Fork" button near the top right and fork it to your own user space
- Now you could either `git clone` your fork to isp02, put in a new file, then `git add => git commit` => ``git push`; OR you could click "Add file" in the GitHub web interface and create a new file that way
- Navigate again to the "Pull requests" tab and click to create a "New pull request"
- Make sure the original repo (jagaither) is set as the base and your fork is set as the head
- Create the pull request and provide enough detail for the repository owner (jagaither) to know whether he should accept your pull request or not

Consult the documentation in the base repository (if documentation exists) and look out for general guidance for contributors. If you develop a new feature and it is merged back into the base, you can generally just delete your fork.

1.7.5 Other Considerations

Most repos will also contain a few standard files in the top directory, including:

README.md: The landing page of your repository on GitHub will display the contents of README.md, if it exists. This is a good place to describe your project and list the appropriate citations. *Please note that all of your homeworks, midterm, and final will require READMEs.*

LICENSE.txt: See if your repository needs a [license](#).

.gitignore: Tells Git which files and directories to ignore when you make a commit.

1.7.6 Additional Resources

- Some of the materials in this module were based on [Software Carpentry DOI: 10.5281/zenodo.57467](#).
- [GitHub Glossary](#)
- [About Branches](#)
- [About Pull Requests](#)
- [About Licenses](#)
- [GitFlow Model](#)
- [More on different git workflows](#)

UNIT 2: WORKING WITH COMMON DATA FORMATS

Please make sure you meet the following prerequisites before continuing past Unit 1:

1. You can log in to the class server, isp02.tacc.utexas.edu
2. You have a strong working knowledge of [Linux commands](#)
3. You have a strong working knowledge of the basic [elements of Python3](#)
4. You have a strong working knowledge of [version control with Git](#)

In Unit 2, we will begin to explore and work hands-on with essential data formats that we will use throughout the course. We will primarily be using Python3 to interact with and manipulate the data. Sample data sets for illustration purposes will be downloaded from various sources, including [NASA's open data site](#).

2.1 Working with JSON

In this hands-on module, we will learn how to work with the JSON data format. JSON (JavaScript Object Notation) is a powerful, flexible, and lightweight data format that we see a lot throughout this course, especially when working with web apps and REST APIs.

After going through this module, students should be able to:

- Identify and write valid JSON
- Read JSON into an object in a Python3 script
- Loop over and work with elements in a JSON object
- Write JSON to file from a Python3 script

2.1.1 JSON Basics

Analogous to Python3 dictionaries, JSON is typically composed of key:value pairs. The universality of this data structure makes it ideal for exchanging information between programs written in different languages and web apps. A simple, valid JSON object may look like this:

```
{  
  "key1": "value1",  
  "key2": "value2"  
}
```

Although less common in this course, simple arrays of information (analogous to Python3 lists) are also valid JSON:

```
[  
  "thing1", "thing2", "thing3"  
]
```

JSON offers a lot of flexibility on the placement of white space and newline characters. Types can also be mixed together, forming complex data structures:

```
{  
  "department": "COE",  
  "number": 332,  
  "name": "Software Engineering and Design",  
  "inperson": true,  
  "finalgroups": null,  
  "instructors": ["Joe", "Charlie", "Joe"],  
  "prerequisites": [  
    {"course": "COE 322", "instructor": "Victor"},  
    {"course": "SDS 322", "instructor": "Victor"}  
  ]  
}
```

On the class server, navigate to your home directory and make a new folder for this module:

```
[local]$ ssh username@isp02.tacc.utexas.edu  
(enter password)  
(enter token)  
[isp02]$ cd coe-332/  
[isp02]$ mkdir working-with-json && cd working-with-json
```

Download this sample JSON files into that folder using the wget command, or click [this link](#) and cut and paste the contents into a file called `Meteorite_Landings.json`:

```
[isp02]$ wget https://raw.githubusercontent.com/TACC/coe-332-sp22/main/docs/unit02/  
↪sample-data/Meteorite_Landings.json
```

Note: The Meteorite Landing data is adapted from a data set provided by The Meteoritical Society here: <https://data.nasa.gov/Space-Science/Meteorite-Landings/gh4g-9sfh>

EXERCISE

Plug this file (or some of the above samples) into an online JSON validator (e.g. [JSONLint](#)). Try making manual changes to some of the entries to see what breaks the JSON format.

2.1.2 Read JSON into a Python3 Script

The `json` Python3 library is part of the Python3 Standard Library, meaning it can be imported without having to be installed by `pip`. Start editing a new Python3 script using your method of choice:

```
[isp02]$ vim json_ex.py
```

Warning: Do not name your Python3 script “`json.py`”. If you `import json` when there is a script called “`json.py`” in the same folder, it will import that instead of the actual `json` library.

The code you need to read in the JSON file of state names and abbreviations into a Python3 object is:

```
1 import json
2
3 with open('Meteorite_Landings.json', 'r') as f:
4     ml_data = json.load(f)
```

Only three simple lines! We `import json` from the standard library so that we can work with the `json` class. We use the safe `with open...` statement to open the file we downloaded read-only into a filehandle called `f`. Finally, we use the `load()` method of the `json` class to load the contents of the JSON file into our new `ml_data` object.

EXERCISE

Try out some of these calls to the `type()` function on the new `ml_data` object that you loaded. Also `print()` each of these as necessary to be sure you know what each is. Be able to explain the output of each call to `type()` and `print()`.

```
1 import json
2
3 with open('Meteorite_Landings.json', 'r') as f:
4     ml_data = json.load(f)
5
6 type(ml_data)
7 type(ml_data['meteorite_landings'])
8 type(ml_data['meteorite_landings'][0])
9 type(ml_data['meteorite_landings'][0]['name'])
10
11 print(ml_data)
12 print(ml_data['meteorite_landings'])
13 print(ml_data['meteorite_landings'][0])
14 print(ml_data['meteorite_landings'][0]['name'])
```

Tip: Consider doing this in the Python3 interpreter’s interactive mode instead of in a script.

2.1.3 Work with JSON Data

As we have seen, the JSON object we loaded contains meteorite landing data including names, ids, classes, masses, latitudes, and longitudes. Let's write a few functions to help us explore the data.

First, write a function to calculate the average mass of all meteorites in the data set. Call that function, and have it print the average mass to screen.

```
1 import json
2
3 def compute_average_mass(a_list_of_dicts, a_key_string):
4     total_mass = 0.
5     for i in range(len(a_list_of_dicts)):
6         total_mass += float(a_list_of_dicts[i][a_key_string])
7     return (total_mass / len(a_list_of_dicts))
8
9 with open('Meteorite_Landings.json', 'r') as f:
10     ml_data = json.load(f)
11
12 print(compute_average_mass(ml_data['meteorite_landings'], 'mass (g)'))
```

Next, write a function to check where on the globe the meteorite landing site is located. We need to check whether it is Northern or Southern hemisphere, and whether it is Western or Eastern hemisphere.

```
1 import json
2
3 def compute_average_mass(a_list_of_dicts, a_key_string):
4     total_mass = 0.
5     for i in range(len(a_list_of_dicts)):
6         total_mass += float(a_list_of_dicts[i][a_key_string])
7     return (total_mass / len(a_list_of_dicts))
8
9 def check_hemisphere(latitude: float, longitude: float) -> str:    # type hints
10     location = ''
11     if (latitude > 0):
12         location = 'Northern'
13     else:
14         location = 'Southern'
15     if (longitude > 0):
16         location = f'{location} & Eastern'
17     else:
18         location = f'{location} & Western'
19     return(location)
20
21 with open('Meteorite_Landings.json', 'r') as f:
22     ml_data = json.load(f)
23
24 print(compute_average_mass(ml_data['meteorite_landings'], 'mass (g)'))
25
26 for row in ml_data['meteorite_landings']:
27     print(check_hemisphere(float(row['reclat']), float(row['reclong'])))
```

Note: Type hints in function definitions indicate what types are expected as input and output of a function, but no

checking actually happens at runtime. Think of them as documentation or annotations.

Tip: Check out Python3 ternary operators to make your if/else conditionals shorter, but perhaps a little less intuitive to read.

```
def check_hemisphere(lat, lon):
    location = 'Northern' if (lat > 0) else 'Southern'
    location = f'{location} & Eastern' if (lon > 0) else f'{location} & Western'
    return(location)
```

EXERCISE

Write a third function to count how many of each 'class' of meteorite there is in the list. The output should look something like:

```
type, number
H, 1
H4, 2
L6, 6
...etc
```

2.1.4 Write JSON to File

Finally, in a new script, we will create an object that we can write to a new JSON file.

```
1 import json
2
3 data = {}
4 data['class'] = 'COE332'
5 data['title'] = 'Software Engineering and Design'
6 data['subjects'] = []
7 data['subjects'].append( {'unit': 1, 'topic': ['linux', 'python3', 'git']} )
8 data['subjects'].append( {'unit': 2, 'topic': ['json', 'csv', 'xml', 'yaml']} )
9
10 with open('class.json', 'w') as out:
11     json.dump(data, out, indent=2)
```

Notice that most of the code in the script above was simply assembling a normal Python3 dictionary. The `json.dump()` method only requires two arguments - the object that should be written to file, and the filehandle. The `indent=2` argument is optional, but it makes the output file look a little nicer and easier to read.

Inspect the output file and paste the contents into an online JSON validator.

EXERCISE

Write a new Python3 script to read in `Meteorite_Landings.json`, convert the ids, masses, latitudes, and longitudes to floats, then save it as a new JSON file called `Meteorite_Landings_updated.json`. Compare them side by side to make sure you can see and understand the difference.

2.1.5 Additional Resources

- [Reference for the JSON library](#)
- [Validate JSON with JSONLint](#)
- [Meteorite Landings Data](#)

2.2 CSV Reference

This reference guide is designed to introduce you to CSV format. CSV (Comma Separated Value) is probably the most intuitive and most human-readable data format we will encounter. Many spreadsheet tools (like Excel) can read and write CSV, and many public data sets we encounter will be stored in CSV. However, there is no data typing, no support for complex data structures, and it lacks in versatility.

2.2.1 CSV Basics

CSV is usually used for simple, tabular data and generally cannot be used to represent some of the more complex data structures we will be exposed to in this class. However, CSV is a very common data format, and you will often encounter tabular CSV data that you want to transform into **a list of dictionaries**.

For example, consider the following valid CSV file with one header line followed by three record lines:

```
firstname,lastname,ID
joe,allen,123
charlie,day,456
joe,stubbs,789
```

The same CSV file could be represented by a **list of dictionaries** where the headers are used as the keys, and the records are used as the values:

```
[
  {
    "firstname": "joe",
    "lastname": "allen",
    "id": 123
  },
  {
    "firstname": "charlie",
    "lastname": "dey",
    "id": 456
  },
  {
    "firstname": "joe",
    "lastname": "stubbs",
    "id": 789
  }
]
```

(continues on next page)

(continued from previous page)

```
}  
]
```

As a slight modification of the above, you will often see that list of dictionaries stored as a **value** and assigned to a **key** with a descriptive name. We will see this exact type of data structure many times this semester:

```
{  
  "instructors": [  
    {  
      "firstname": "joe",  
      "lastname": "allen",  
      "id": 123  
    },  
    {  
      "firstname": "charlie",  
      "lastname": "dey",  
      "id": 456  
    },  
    {  
      "firstname": "joe",  
      "lastname": "stubbs",  
      "id": 789  
    }  
  ]  
}
```

The latest specification for CSV provides the following guidelines that “seem to be followed by most implementations” of CSV:

1. Each record is located on a separate line with a line break
2. The last record in the file may or may not have a line break
3. There maybe an optional header line appearing as the first line of the file with the same format as normal record lines
4. Within the header and each record, there may be one or more fields, separated by commas. Each line should contain the same number of fields throughout the file
5. Each field may or may not be enclosed in double quotes
6. Fields containing line breaks, double quotes, and commas should be enclosed in double-quotes
7. If double-quotes are used to enclose fields, then a double-quote appearing inside a field must be escaped by preceding it with another double quote

Source: <https://datatracker.ietf.org/doc/html/rfc4180>

Keep an eye out for similar formats, too. For example .tsv files generally are the exact same thing as .csv files except tab-delimited rather than comma-delimited.

Note: Check out the list of meteorite landing sites we worked with in the JSON section, but now in CSV format [here](#).

2.2.2 Read CSV from File

Imagine you have a CSV file with headers, e.g.:

```
name,id,recclass,mass (g),reclat,reclong,GeoLocation
Ruiz,10001,L5,21,50.775,6.08333,"(50.775, 6.08333)"
Beeler,10002,H6,720,56.18333,10.23333,"(56.18333, 10.23333)"
Brock,10003,EH4,107000,54.21667,-113,"(54.21667, -113.0)"
Hillebrand,10004,Acapulcoite,1914,16.88333,-99.9,"(16.88333, -99.9)"
Mitchell,10005,L6,780,-33.16667,-64.95,"(-33.16667, -64.95)"
... etc
```

To read it into a dictionary object, use the csv module that is part of the Python3 standard library:

```
1 import csv
2
3 data = {}
4 data['meteorite_landings'] = []
5
6 with open('Meteorite_Landings.csv', 'r') as f:
7     reader = csv.DictReader(f)
8     for row in reader:
9         data['meteorite_landings'].append(dict(row))
```

In the above case, each row of the CSV file is iterated one by one. The keys from the header line are assigned values from the subsequent lines, then they are appended to a list of dictionaries in the data data structure.

2.2.3 Write CSV to File

Given the same data structure as in the previous example (a list of dictionaries where each dictionary has identical keys), to write that object to a CSV file, use the following:

```
1 import csv
2 import json
3
4 data = {}
5
6 with open('Meteorite_Landings.json', 'r') as f:
7     data = json.load(f)
8
9 with open('Meteorite_Landings.csv', 'w') as o:
10     csv_dict_writer = csv.DictWriter(o, data['meteorite_landings'][0].keys())
11     csv_dict_writer.writeheader()
12     csv_dict_writer.writerows(data['meteorite_landings'])
```

2.2.4 Additional Resources

- [CSV Basics](#)
- [CSV Module in the Python Standard Library](#)

2.3 XML Reference

This reference guide is designed to introduce you to XML syntax. XML (Extensible Markup Language) was designed as a way to store, transmit, and reconstruct arbitrary data sets, particularly over the internet. You'll sometimes see it as standalone documents, or as streaming data without a definite endpoint. For this class, we will focus on the standalone documents / datasets.

2.3.1 XML Basics

Much like JSON, XML can be used to store arbitrary data sets in a dictionary-esque format of key:value pairs. However, there are a few key differences between XML and JSON:

1. XML is typeless, essentially everything is a string
2. XML documents support comments
3. XML documents require exactly one “root” element, so the data structure of a **dictionary with one key, whose value is a list of dictionaries** needs a slight modification in order to be valid JSON (as we will see below)

A valid XML document takes the form:

```
<data>
  <key1>value1</key1>
  <key2>value2</key2>
</data>
```

Keys always come as a pair of start-tag (<key1>) and end-tag (</key1>) markups surrounding a plain text value (value1). In the above example, the root-level key data was required to make this valid XML. Without it, the keys key1 and key2 would be at the same root level, which is invalid. The same data shown above in JSON format would appear as:

```
{
  "data": {
    "key1": "value1",
    "key2": "value2"
  }
}
```

Things get a little tricky with our favorite data structure, a **dictionary with one key, whose value is a list of dictionaries**. This is because of the way XML represents lists of dictionaries. Consider the following snippet of JSON data:

```
{
  "instructors": [
    {
      "firstname": "joe",
      "lastname": "allen",
      "id": 123
    },
    ...
  ]
}
```

(continues on next page)

(continued from previous page)

```

{
  "firstname": "charlie",
  "lastname": "dey",
  "id": 456
},
{
  "firstname": "joe",
  "lastname": "stubbs",
  "id": 789
}
]
}

```

If we try to translate this directly to XML, it would take the form:

```

<instructors>
  <firstname>joe</firstname>
  <lastname>allen</lastname>
  <id>123</id>
</instructors>
<instructors>
  <firstname>charlie</firstname>
  <lastname>dey</lastname>
  <id>456</id>
</instructors>
<instructors>
  <firstname>joe</firstname>
  <lastname>stubbs</lastname>
  <id>789</id>
</instructors>

```

The instructors key appears multiple times at the root level, once for each element in the list. In XML, you cannot have multiple roots, even if it is the same root repeated more than once. You need exactly one root only. A simple trick to fix this is to create a new dictionary with one key, e.g. “data”, whose value is the other dictionary. Doing so would slightly change the XML to a valid format:

```

<data>
  <instructors>
    <firstname>joe</firstname>
    <lastname>allen</lastname>
    <id>123</id>
  </instructors>
  <instructors>
    <firstname>charlie</firstname>
    <lastname>dey</lastname>
    <id>456</id>
  </instructors>
  <instructors>
    <firstname>joe</firstname>
    <lastname>stubbs</lastname>
    <id>789</id>
  </instructors>
</data>

```


Note: Check out the list of meteorite landing sites we worked with in the JSON section, but now in XML format [here](#).

2.3.2 Read XML from File

Here we will focus on the “document object model” for parsing XML, which means we will read in one XML document and parse the entire thing as a whole. (This works for reasonably small files that can fit in memory).

Note that the Python3 standard library has an XML module, but it does not have a method for transforming XML objects to dictionaries. Since most of what we do in this class uses JSON and dictionaries, let’s instead use the `xmltodict` Python module which works directly in dictionary space.

Warning: Install the `xmltodict` library before proceeding:

```
[isp02]$ pip3 install --user xmltodict
```

You can read in an XML file (e.g. the Meteorite Landings data linked above) and store it as a dictionary as follows:

```
import xmltodict

with open('Meteorite_Landings.xml', 'r') as f:
    data = xmltodict.parse(f.read())
```

Then to access the data within that dictionary, remember to include an extra key for the root-level, which we added in to make valid XML. For example, you could call out the first meteorite in the list with the following:

```
print(data['data']['meteorite_landings'][0])
```

Note: The original Meteorite Landings data had the key `mass` (g). The open and close parentheses () are invalid characters when it comes to XML tags. The data linked above was modified to use the key `mass_g` instead. Don’t be surprised when working with datasets if you have to make manual modifications to the data in order to make it valid in a particular format.

2.3.3 Write XML to File

As mentioned above, a dictionary must have exactly one “root” element in order to write valid XML. The following example below assembles a dictionary with multiple keys at the root level (“class”, “title”, “subjects”). In fact the following code will yield an error:

```
1 import xmltodict
2
3 data = {}
4 data['class'] = 'COE332'
5 data['title'] = 'Software Engineering and Design'
6 data['subjects'] = []
7 data['subjects'].append( {'unit': 1, 'topic': ['linux', 'python3', 'git']} )
8 data['subjects'].append( {'unit': 2, 'topic': ['json', 'csv', 'xml', 'yaml']} )
9
```

(continues on next page)

(continued from previous page)

```
10 with open('class.xml', 'w') as o:  
11     o.write(xmltodict.unparse(data, pretty=True))
```

Error:

```
ValueError: Document must have exactly one root.
```

To get this to work, you need to modify the above script to create a new dictionary, e.g. “root”, with exactly one key, e.g. “data”, whose value is the entire data dictionary:

```
1 import xmltodict  
2  
3 data = {}  
4 data['class'] = 'COE332'  
5 data['title'] = 'Software Engineering and Design'  
6 data['subjects'] = []  
7 data['subjects'].append( {'unit': 1, 'topic': ['linux', 'python3', 'git']} )  
8 data['subjects'].append( {'unit': 2, 'topic': ['json', 'csv', 'xml', 'yaml']} )  
9  
10 root = {}  
11 root['data'] = data  
12  
13 with open('class.xml', 'w') as o:  
14     o.write(xmltodict.unparse(root, pretty=True))
```

Output:

```
<data>  
  <class>COE332</class>  
  <title>Software Engineering and Design</title>  
  <subjects>  
    <unit>1</unit>  
    <topic>linux</topic>  
    <topic>python3</topic>  
    <topic>git</topic>  
  </subjects>  
  <subjects>  
    <unit>2</unit>  
    <topic>json</topic>  
    <topic>csv</topic>  
    <topic>xml</topic>  
    <topic>yaml</topic>  
  </subjects>  
</data>
```

2.3.4 Additional Resources

- [The xmltodict Library](#)
- [XML Linter](#)

2.4 YAML Reference

This reference guide is designed to introduce you to YAML syntax. YAML (YAML Ain't Markup Language) is basically JSON with a couple extra features, and meant to be a little more human readable. We will be using YAML formatted configuration files in the Docker compose and Kubernetes sections, so it is important to become familiar with the syntax.

2.4.1 YAML Basics

YAML syntax is similar to Python dictionaries, and we will usually see them as key:value pairs. Values can include strings, numbers, booleans, null, lists, and other dictionaries.

Previously, we saw a simple JSON object in dictionary form like:

```
{
  "key1": "value1",
  "key2": "value2"
}
```

That same object in YAML looks like:

```
---
key1: value1
key2: value2
...
```

Notice that YAML documents all start with three hyphens on the top line (---), and end with an optional three dots (...) on the last line. Key:value pairs are separated by colons, but consecutive key:value pairs are NOT separated by commas.

We also mentioned that JSON supports list-like structures. YAML does too. So the following valid JSON block:

```
[
  "thing1", "thing2", "thing3"
]
```

Appears like this in YAML:

```
---
- thing1
- thing2
- thing3
...
```

Elements of the same list all appear with a hyphen - at the same indent level.

We previously saw this complex data structure in JSON:

```
{
  "department": "COE",
  "number": 332,
  "name": "Software Engineering and Design",
  "inperson": true,
  "finalgroups": null,
  "instructors": ["Joe", "Charlie", "Joe"],
  "prerequisites": [
    {"course": "COE 322", "instructor": "Victor"},
    {"course": "SDS 322", "instructor": "Victor"}
  ]
}
```

The same structure in YAML is:

```
---
department: COE
number: 332
name: Software Engineering and Design
inperson: true
finalgroups: null           # can also use ~
instructors:
  - Joe
  - Charlie
  - Joe
prerequisites:
  - course: COE 322
    instructor: Victor
  - course: SDS 322
    instructor: Victor
...
```

The whole thing can be considered a dictionary. The key `instructors` contains a value that is a list of names, and the key `prerequisites` contains a value that is a list of dictionaries. Booleans appear as `false` and `true` (lowercase only). Null / empty values appear as `null` or `~`. And, as you can see above, YAML also supports comments starting with a `#`.

One glaring thing that is missing from the YAML file is quotation marks. In general, you don't have to use quotes in YAML. You may use quotes to force a number to be interpreted as a string (e.g. `10` will automatically be interpreted as an integer, but `"10"` will be interpreted as a string).

Note: Check out the list of meteorite landing sites we worked with in the JSON section, but now in YAML format [here](#).

There is a lot more to YAML, most of which we will not use in this course. Just know that YAML files can contain:

- Comments
- Multi-line strings / text blocks
- Multi-word keys
- Complex objects
- Special characters

- Explicitly declared types
- A mechanism to duplicate / inherit values across a document (“anchors”)

If we encounter a need for any of these, we can refer to the [official YAML syntax](#)

2.4.2 Read YAML from File

Warning: There is no YAML interpreter in the Python 3.6 standard library, so we need to install one with pip3:

```
[isp02]$ pip3 install --user pyyaml
```

Given the meteorite landing site data in YAML format, which you can download from [this link](#), load it into a Python3 dictionary object using the following:

```
1 import yaml
2
3 data = {}
4
5 with open('Meteorite_Landings.yaml', 'r') as f:
6     data = yaml.load(f, Loader=yaml.SafeLoader)
```

Very similar to the JSON module, it only requires a few simple lines then you have a dictionary object to work with. The `Loader=yaml.SafeLoader` parameter makes it so no arbitrary Python code is executed when loading in the data - this is typically a good choice for data from untrusted sources.

2.4.3 Write YAML to File

In a new script create a dictionary object that we can write to a new YAML file.

```
1 import yaml
2
3 data = {}
4 data['class'] = 'COE332'
5 data['title'] = 'Software Engineering and Design'
6 data['subjects'] = []
7 data['subjects'].append( {'unit': 1, 'topic': ['linux', 'python3', 'git']} )
8 data['subjects'].append( {'unit': 2, 'topic': ['json', 'csv', 'xml', 'yaml']} )
9
10 with open('class.json', 'w') as o:
11     yaml.dump(data, o)
```

Notice that most of the code in the script above was simply assembling a normal Python3 dictionary. The `json.dump()` method only requires two arguments - the object that should be written to file, and the filehandle.

Inspect the output file and paste the contents into an online YAML validator.

2.4.4 Additional Resources

- [YAML Spec](#)
- [YAML Validator](#)
- [JSON / YAML Converter](#)
- [PyYAML Docs](#)

UNIT 3: BEST PRACTICES IN PYTHON

In Unit 3, we will work on improving our Python3 code-writing habits to make our code easier to read, easier to debug, and easier to test. A little extra effort towards writing good code now will pay off down the line when it comes to developing and testing new features or re-using old code blocks in new applications. Specifically, we will learn and practice topics in code organization, documentation, logging, and unit testing.

3.1 Code Organization

Following standard Python3 code organization practices will make our code easier to read by other developers, and by our future selves who are looking back to see what we did. After going through this module, students should be able to:

- Organize their code into `main()` functions
- Import their functions into other scripts without executing the `main()` block
- Write functions in a generalizable way so they are reusable
- Use a shebang in their Python3 scripts to make them executable

3.1.1 Main Function

In many Python programs, you will find the developer has organized their code into a `main()` function. Then, they will only call the `main()` function if the variable `__name__` is equal to the string `'__main__'`. For example:

```
def main():  
    # the meat of the main function goes here  
  
if __name__ == '__main__':  
    main()
```

If this script is executed on the command line directly, then the internal variable `__name__` will be set to the string `'__main__'`. The conditional evaluates as True and the `main()` function is called.

If this script is instead **imported into another script**, say, to reuse some of the functions defined within, then the internal variable `__name__` will instead be set to the name of the script. Thus, the `main()` function is not called, but other functions defined in this script would be available.

Consider the script we wrote in the previous unit for analyzing the Meteorite Landings JSON file (called `ml_data_analysis.py`):

```

1 import json
2
3 def compute_average_mass(a_list_of_dicts, a_key_string):
4     total_mass = 0.
5     for item in a_list_of_dicts:
6         total_mass += float(item[a_key_string])
7     return(total_mass / len(a_list_of_dicts) )
8
9 def check_hemisphere(latitude, longitude):
10    location = 'Northern' if (latitude > 0) else 'Southern'
11    location = f'{location} & Eastern' if (longitude > 0) else f'{location} & Western'
12    return(location)
13
14 with open('Meteorite_Landings.json', 'r') as f:
15     ml_data = json.load(f)
16
17 print(compute_average_mass(ml_data['meteorite_landings'], 'mass (g)' ))
18
19 for row in ml_data['meteorite_landings']:
20     print(check_hemisphere(float(row['reclat']), float(row['reclong'])))

```

To reorganize this code, we would put the file read operation and the two function calls into a main function:

```

1 import json
2
3 def compute_average_mass(a_list_of_dicts, a_key_string):
4     total_mass = 0.
5     for item in a_list_of_dicts:
6         total_mass += float(item[a_key_string])
7     return(total_mass / len(a_list_of_dicts) )
8
9 def check_hemisphere(latitude, longitude):
10    location = 'Northern' if (latitude > 0) else 'Southern'
11    location = f'{location} & Eastern' if (longitude > 0) else f'{location} & Western'
12    return(location)
13
14 def main():          # notice the below lines are now indented
15     with open('Meteorite_Landings.json', 'r') as f:
16         ml_data = json.load(f)
17
18     print(compute_average_mass(ml_data['meteorite_landings'], 'mass (g)' ))
19
20     for row in ml_data['meteorite_landings']:
21         print(check_hemisphere(float(row['reclat']), float(row['reclong'])))
22
23 if __name__ == '__main__':
24     main()

```

If this code is imported into another Python3 script, that other script will have access to the `compute_average_mass()` and `check_hemisphere()` functions, but it will not execute the code in the `main()` function.

EXERCISE

Write a new script to import the above code, assuming that above code is saved in a file called `ml_data_analysis.py`:

```
1 import ml_data_analysis      # assumes it is in this directory, or installed in known_
   ↪ location
2
3 print(ml_data_analysis.check_hemisphere(35.0, 70.0))
4 print(ml_data_analysis.check_hemisphere(-35.0, -70.0))
```

Try executing this new script with and without protecting the imported code in a `main()` function. How do the outputs differ?

Tip: The main function does not have to be called literally `main()`. But, if someone else is reading your code, calling it `main()` will certainly help orient the reader.

3.1.2 Generalizing Functions

A good habit to get into while writing functions is to write them in a *generalizable* way. This means writing them in such a way that they can be used for multiple purposes or in multiple applications. The trick is to try to think ahead about how else you might use the function, think about what form the input data takes, and try not to hardcode indices or variable names.

`compute_average_mass`

In our `compute_average_mass` function, we knew we needed to send it *something*, and we knew it needed to return an average mass. The main question was what form should the input take?

```
def compute_average_mass( xyz ):
    # do some computation
    return(average_mass)
```

We could have just sent the function the entire dictionary data structure, then have it parse the data to get masses out. But if we did that, we would also need to hardcode the name of the main key 'meteorite_landings' as well as the name of the key referring to the masses 'mass (g)'.

```
# BAD
def compute_average_mass( a_dictionary ):
    total_mass = 0.
    for item in a_dictionary['meteorite_landings']:
        total_mass += float(item['mass (g)'])
    return(total_mass / len(a_dictionary['meteorite_landings']) )

print(compute_average_mass(ml_data))
```

Since we will be working with lists of dictionaries most frequently in this class, it makes more sense to send it a list of dictionaries data structure and the name of the key to extract.

```
# GOOD
def compute_average_mass(a_list_of_dicts, a_key_string):
    total_mass = 0.
```

(continues on next page)

(continued from previous page)

```

    for item in a_list_of_dicts:
        total_mass += float(item[a_key_string])
    return(total_mass / len(a_list_of_dicts) )

print(compute_average_mass(ml_data['meteorite_landings'] , 'mass (g)' ))

```

check_hemisphere

The check_hemisphere function is very similar - we send it *something* and it returns (or prints) a string.

```

def check_hemisphere( xyz )
    # run through some conditionals
    return(location)

```

Here we could have also sent a list of dictionaries along with the names of two keys representing the latitude and longitude. That would have been ok, and would have worked for most of the data structures we use in this class.

```

# NOT TERRIBLE
def check_hemisphere(a_list_of_dicts, lat_key, long_key):
    for item in a_list_of_dicts:
        location = 'Northern' if (float(item[lat_key]) > 0) else 'Southern'
        location = f'{location} & Eastern' if (float(item[long_key]) > 0) else f'
↪{location} & Western'
        print(location)
    return

check_hemisphere(ml_data['meteorite_landings'], 'reclat', 'reclong')

```

However, to make it even more generalizable, we could abstract one layer further and just send it two floats: latitude and longitude. That would make the function useful for our list of dictionaries data structure, and for one-off checks given just a pair of floats:

```

# BETTER
def check_hemisphere(latitude, longitude):
    location = 'Northern' if (latitude > 0) else 'Southern'
    location = f'{location} & Eastern' if (longitude > 0) else f'{location} & Western'
    return(location)

for row in ml_data['meteorite_landings']:
    print(check_hemisphere(float(row['reclat']), float(row['reclong'])))

```

EXERCISE

Write a new function to count how many of each ‘class’ of meteorite there is in the list. The output should look something like:

```

type, number
H, 1
H4, 2
L6, 6
...etc

```

Consider carefully what inputs you are sending to the function. How can you write it in a generalizable way?

3.1.3 Shebang

A “shebang” is a line at the top of your script that defines what interpreter should be used to run the script when treated as a standalone executable. You will often see these used in Python, Perl, Bash, C shell, and a number of other scripting languages. In our case, we want to use the following shebang, which should appear on the first line of our Python3 scripts:

```
#!/usr/bin/env python3
```

The `env` command simply figures out which version of `python3` appears first in your path, and uses that to execute the script. We usually use that form instead of, e.g., `#!/usr/bin/python3.6` because the location of the Python3 executable may differ from machine to machine, whereas the location of `env` will not.

Next, you also need to make the script executable using the Linux command `chmod`:

```
[isp02]$ chmod u+x ml_data_analysis.py
```

That enables you to call the Python3 code within as a standalone executable without invoking the interpreter on the command line:

```
[isp02]$ ./ml_data_analysis.py
```

This is helpful to lock in a Python version (e.g. Python3) for a script that may be executed on multiple different machines or in various environments.

3.1.4 Other Tips

As our Python3 scripts become longer and more complex, we should put more thought into how the different contents of the script are ordered. As a rule of thumb, try to organize the different sections of your Python3 code into this order:

```
# Shebang

# Imports

# Global variables / constants

# Class definitions

# Function definitions

# Main function definition

# Call to main function
```

Other general tips for writing code that is easy to read can be found in the [PEP 8 Style Guide](#), including:

- Use four spaces per indentation level (no tabs)
- Limit lines to 80 characters, wrap and indent where needed
- Avoid extraneous whitespace unless it improves readability
- Be consistent with naming variables and functions

- Classes are usually `CapitalWords`
- Constants are usually `ALL_CAPS`
- Functions and variables are usually `lowercase_with_underscores`
- Consistency is the key
- Use functions to improve organization and reduce redundancy
- Document and comment your code

Note: Beyond individual Python3 scripts, there is a lot more to learn about organizing *projects* which may consist of many files. We will get into this later in the semester.

3.1.5 Additional Resources

- [PEP 8 Style Guide](#)

3.2 Documentation

As we have probably all heard before, good documentation is almost as important (if not equally as important) as good code itself. You may have written some elegant and powerful code to solve all your problems today, but weeks or months today does that code become functionally useless if you forget what it does or how to call it? Python3 users have a special built-in tool at their disposal called *docstrings* that make documenting functions easy. After going through this module, students should be able to:

- Write well-crafted docstrings for all functions
- Add type hints to function definitions
- Write effective READMEs for a project

3.2.1 Docstrings

Docstrings are special strings that appear immediately following function definitions in our code. They should be surrounded by three double-quotation marks on each side, and they may span multiple lines. For example:

```
def a_function():  
    """  
    This is a docstring.  
    """  
    # code goes here  
    return
```

The above is a valid docstring, but it is not a very helpful docstring. When you write docstrings, at a minimum try to include the following sections:

1. A short description of the purpose of the function
2. A list of arguments, including type
3. A list of returned values, including type

A better template for a docstring (based on the [Google Style Guide](#)) might look like:

```
def a_function(arg1, arg2):
    """
    This function does XYZ.

    Args:
        arg1 (type): Define what is expected for arg1.
        arg2 (type): Define what is expected for arg2.

    Returns:
        result (type): Define what is expected for result.
    """
    # code goes here
    return(result)
```

The description should be succinct, yet complete. Arguments should be listed by name and the expected type (e.g., bool, float, str, etc) should be stated. And the return result(s) should be listed along with the expected type(s).

Let's look at one more example using a real function:

```
def add_and_square(num1, num2):
    """
    Given two numbers, this function will first add them together, then square the sum
    and return the result.

    Args:
        num1 (float): The first number.
        num2 (float): The second number.

    Returns:
        result (float): The square of the sum of input arguments.
    """
    result = (num1+num2)**2
    return(result)
```

Note: Notice above we are using more-or-less complete sentences with proper grammar.

Next, let's add docstrings to our `ml_data_analysis.py` code we have been working on:

```
1  #!/usr/bin/env python3
2  import json
3
4  def compute_average_mass(a_list_of_dicts, a_key_string):
5      """
6      Iterates through a list of dictionaries, pulling out values associated with
7      a given key. Returns the average of those values.
8
9      Args:
10         a_list_of_dicts (list): A list of dictionaries, each dict should have the
11                                same set of keys.
12         a_key_string (string): A key that appears in each dictionary associated
13                               with the desired value (will enforce float type).
14
```

(continues on next page)

(continued from previous page)

```

15     Returns:
16         result (float): Average value.
17     """
18     total_mass = 0.
19     for item in a_list_of_dicts:
20         total_mass += float(item[a_key_string])
21     return(total_mass / len(a_list_of_dicts) )
22
23 def check_hemisphere(latitude, longitude):
24     """
25     Given latitude and longitude in decimal notation, returns which hemispheres
26     those coordinates land in.
27
28     Args:
29         latitude (float): Latitude in decimal notation.
30         longitude (float): Longitude in decimal notation.
31
32     Returns:
33         location (string): Short string listing two hemispheres.
34     """
35     location = 'Northern' if (latitude > 0) else 'Southern'
36     location = f'{location} & Eastern' if (longitude > 0) else f'{location} & Western'
37     return(location)
38
39 def count_classes(a_list_of_dicts, a_key_string):
40     """
41     ???
42     """
43     classes_observed = {}
44     for item in a_list_of_dicts:
45         if item[a_key_string] in classes_observed:
46             classes_observed[item[a_key_string]] += 1
47         else:
48             classes_observed[item[a_key_string]] = 1
49     return(classes_observed)
50
51 def main():
52     with open('Meteorite_Landings.json', 'r') as f:
53         ml_data = json.load(f)
54
55     print(compute_average_mass(ml_data['meteorite_landings'], 'mass (g)'))
56
57     for row in ml_data['meteorite_landings']:
58         print(check_hemisphere(float(row['reclat']), float(row['reclong'])))
59
60     print(count_classes(ml_data['meteorite_landings'], 'recclass'))
61
62 if __name__ == '__main__':
63     main()

```

In general, your main() function usually does not need a docstring. It is good habit to write the main() function simply and clearly enough that it is self explanatory, with perhaps a few comments to help. If you do add a docstring to the main() function, you may write a few short summary sentences but omit the Args and Returns sections.

EXERCISE

Write the missing docstring for the `count_classes()` function above.

EXERCISE

Open up the Python3 interactive interpreter. Import your `ml_data_analysis.py` methods. Use the commands `dir()` and `help()` to find and read the docstrings that you wrote.

3.2.2 Type Hints

Type hints in function definitions indicate what types are expected as input and output of a function. No checking actually happens at runtime, so if you send the wrong type of data as an argument, the type hint itself won't cause it to return an error. Think of type hints simply as documentation or annotations to help the reader understand how to use a function.

Warning: In the code blocks below, we omit docstrings for brevity only. Please keep including docstrings in your code.

Type hints should take form:

```
def a_function(arg_name: arg_type) -> return_type:
    # code goes here
    return(result)
```

In the above example, we are providing a single argument called `arg_name` that should be of type `arg_type`. The expected return value should be `return_type`. Let's look at an example using a real function:

```
def add_and_square(num1: float, num2: float) -> float:
    result = (num1+num2)**2
    return(result)
```

Next, add type hints to the function definitions of the `ml_data_analysis.py` script (only showing snippets below):

```
from typing import List
def compute_average_mass(a_list_of_dicts: List[dict], a_key_string: str) -> float:
```

```
def check_hemisphere(latitude: float, longitude: float) -> float:
```

```
def count_classes(a_list_of_dicts, a_key_string): # what about this one?
```

In the first example above we need to include the line `from typing import List` to get access to a special object called `List` (capital `L`). We use that to object to not only hint that `a_list_of_dicts` should be a list, but it also includes information about what type of list we expect - in this case it is a list of dictionaries. In Python3.6 you cannot do this with the normal `list` (lowercase `l`*) object.

Although Python3 does not check or enforce types at run time, there are other tools that make use of type hints to check types at the time of development. For example, some IDEs (including PyCharm) will evaluate type hints as you write code and provide an alert if you call a function in a way other than what the type hint suggests. In addition, there are Python3 libraries like `Mypy` that can wrap your Python3 programs and check / evaluate type hints as you go, provided errors where types don't match.

Warning: Be aware that there is some redundancy in the information contained in type hints and in the docstrings. Be careful not to let them get out of sync as your code evolves.

3.2.3 README

A README file should be included at the top level of every coding project you work on. Websites like GitHub will automatically look for README files and render them directly in the web interface. Markdown is probably the most common syntax people use to write READMEs. It is very easy to create headers, code blocks, tables, text emphases, and other fancy renderings to make the README pleasant and easy to read.

Note: In this class we ask you to include READMEs in each of your homework folders on GitHub. Each homework is essentially a standalone project, so a dedicated README for each is warranted.

At a minimum, plan to include the following sections in all of your READMEs:

- Title: a descriptive, self-explanatory title for the project.
- Description: a high-level description of the project that informs the reader what the code does, why it exists, what problem it solves, etc.
- Installation: As we advance into the semester our code bases will become more complex with more moving parts. Eventually we will need to start providing detailed instructions about getting the project working plus any requirements.
- Usage: The key here is **examples!** Show code blocks of what it looks like to execute the code from start to finish. Describe what output is expected and how it should be interpreted.

Other general advice includes:

- Use proper grammar and more-or-less complete sentences.
- Use headers, code blocks, and text emphases (e.g. bold, italics) to make the document readable. There are plenty of tools to preview Markdown before committing to GitHub, so plan to go through several cycles of editing -> previewing to make your README look nice.
- Be prepared to include other information about authors, acknowledgements, and licenses in the READMEs as appropriate
- Spend some time browsing GitHub and look for READMEs of other popular projects. There is no one correct way to write a README, but there are plenty of wrong ways.

Remember, the README is your chance to document for yourself and explain to others why the project is important, what the code is, and how to use it / interpret the outputs. The advice above is general advice, but it is not one-size-fits-all. Every project is different and ultimately your README may include other sections or organization schemes that are unique to your project.

3.2.4 Additional Resources

- [Google Style Guide for docstrings](#)
- [Type hints spec](#)
- [Mypy project](#)
- [Markdown syntax](#)
- [Tips on writing a good README](#)

3.3 Logging

As your Python3 applications grow in complexity, so too does your need to be able to track what is going on in the code during run time. *Logging* can be used to track arbitrary events as the code runs. As we will see, logging will be useful to keep track of the progression of your code, and it will also be useful to help us track what parts of our code are causing errors. This is especially useful when working with multiple applications across distributed systems. After going through this module, students should be able to:

- Import the logging module into their Python3 scripts
- Set log level to either DEBUG, INFO, WARNING, ERROR, or CRITICAL
- Write appropriate log messages for each log level
- Read output logs and track warnings / errors back to specific spots in their code

3.3.1 Log Levels

We probably all have used (use?) arbitrary print statements in our code as a means to debug errors. Yes, there is a better way! The Python3 logging module, part of the Standard Library, provides functions for reporting different types of events that occur during run time. Save print statements for printing out the normal things that the code is supposed to display, and use exceptions to interrupt the code when it encounters errors. Use logging for everything else, including:

- Printing detailed information about normal things that are supposed to occur, but should not be in the standard output
- Printing warnings about particular run time events
- Printing when an error has occurred but was suppressed by, e.g., an error handler.

Log levels are used to distinguish between the severity or importance of the different events. Using different log levels, you can always leave the log statements which print useful information in your code, but toggle them on and off depending on which level of severity you want to monitor. The standard log levels and their purposes are:

- **DEBUG:** Detailed information, typically of interest only when diagnosing problems
- **INFO:** Confirmation that things are working as expected.
- **WARNING:** An indication that something unexpected happened, or indicative of some problem in the near future (e.g. 'disk space low'). The software is still working as expected.
- **ERROR:** Due to a more serious problem, the software has not been able to perform some function.
- **CRITICAL:** A serious error, indicating that the program itself may be unable to continue running.

(Source: Python Docs <https://docs.python.org/3/howto/logging.html>)

3.3.2 Initialize Logging

Let's work through an example. Add the following lines to a script called, e.g. `log_test.py`:

```
1 import logging
2 logging.basicConfig()      # configs the logging instance
3
4 logging.debug('This is a DEBUG message')
5 logging.info('This is an INFO message')
6 logging.warning('This is a WARNING message')
7 logging.error('This is an ERROR message')
8 logging.critical('This is a CRITICAL message')
```

Executing that code will output the following messages:

```
[isp02]$ python3 log_test.py
WARNING:root:This is a WARNING message
ERROR:root:This is an ERROR message
CRITICAL:root:This is a CRITICAL message
```

By default, the log level is set to `WARNING`, and only messages that are `WARNING` or higher in level of severity will be printed to screen.

If you set a lower log level, e.g. to `DEBUG`, all levels of log messages will be printed:

```
1 import logging
2 logging.basicConfig(level=logging.DEBUG)
3
4 logging.debug('This is a DEBUG message')
5 logging.info('This is an INFO message')
6 logging.warning('This is a WARNING message')
7 logging.error('This is an ERROR message')
8 logging.critical('This is a CRITICAL message')
```

```
[isp02]$ python3 log_test.py
DEBUG:root:This is a DEBUG message
INFO:root:This is an INFO message
WARNING:root:This is a WARNING message
ERROR:root:This is an ERROR message
CRITICAL:root:This is a CRITICAL message
```

An even better set up would be to pass the desired log level dynamically each time you execute the code. But for now, it will be sufficient to manually edit the `basicConfig()` line if we want to change the log level.

3.3.3 What to Include in a Log

As we work toward systems in which we are running multiple applications distributed over remote systems, it is important to be mindful of what sort of log information will be useful. In particular, it would be a good idea to be able to gather information about:

- **Timestamp:** when the error occurred, also acts as a good reference point when referring to a specific log message
- **Hostname:** what (virtual) machine the error occurred on, as you may have multiple instances of an application running on different machines

- **Locale:** what script and/or what function did the message originate from, helps to pinpoint where the message is coming from

To include some of this information in a log message, we need to specify a little more information in the basic config. We also need to import the `socket` module from the Standard Library so we can grab information about the hostname from the environment. We also will be calling a few other of the logging formatter's pre-defined macros.

```

1 import logging
2 import socket
3 format_str=f'[%(asctime)s {socket.gethostname()}] %(filename)s:%(funcName)s:%(lineno)s -
↳ %(levelname)s: %(message)s'
4 logging.basicConfig(level=logging.DEBUG, format=format_str)
5
6 logging.debug('This is a DEBUG message')
7 logging.info('This is an INFO message')
8 logging.warning('This is a WARNING message')
9 logging.error('This is an ERROR message')
10 logging.critical('This is a CRITICAL message')
```

```

[isp02]$ python3 log_test.py
[2022-02-08 02:22:11,627 isp02.tacc.utexas.edu] log_test.py:<module>:9 - DEBUG: This is_
↳ a DEBUG message
[2022-02-08 02:22:11,627 isp02.tacc.utexas.edu] log_test.py:<module>:10 - INFO: This is_
↳ an INFO message
[2022-02-08 02:22:11,628 isp02.tacc.utexas.edu] log_test.py:<module>:11 - WARNING: This_
↳ is a WARNING message
[2022-02-08 02:22:11,628 isp02.tacc.utexas.edu] log_test.py:<module>:12 - ERROR: This is_
↳ an ERROR message
[2022-02-08 02:22:11,628 isp02.tacc.utexas.edu] log_test.py:<module>:13 - CRITICAL: This_
↳ is a CRITICAL message
```

Later in the semester, most of the work we will do will be containerized. It is a little difficult to retrieve *log files* from inside containers, especially if they crash with an error. An easy work around is to use logging to print to standard out (as above), and those messages will end up in the container logs from which they are easily extracted.

EXERCISE

Given the Meteorite Landings analysis script we have been working on, add some logging throughout the script, focusing on DEBUG and ERROR messages.

Note: Note that docstrings below were shortened to save space.

```

1 #!/usr/bin/env python3
2 import json
3 from typing import List
4
5 def compute_average_mass(a_list_of_dicts: List[dict], a_key_string: str) -> float:
6     """
7     Iterates through a list of dictionaries, pulling out values associated with
8     a given key. Returns the average of those values.
9     """
10    total_mass = 0.
```

(continues on next page)

(continued from previous page)

```
11     for item in a_list_of_dicts:
12         total_mass += float(item[a_key_string])
13     return(total_mass / len(a_list_of_dicts) )
14
15 def check_hemisphere(latitude: float, longitude: float) -> float:
16     """
17     Given latitude and longitude in decimal notation, returns which hemispheres
18     those coordinates land in.
19     """
20     location = 'Northern' if (latitude > 0) else 'Southern'
21     location = f'{location} & Eastern' if (longitude > 0) else f'{location} & Western'
22     return(location)
23
24 def count_classes(a_list_of_dicts: List[dict], a_key_string: str) -> dict:
25     """
26     Iterates through a list of dictionaries, and pulls out the value associated
27     with a given key. Counts the number of times each value occurs in the list of
28     dictionaries and returns the result.
29     """
30     classes_observed = {}
31     for item in a_list_of_dicts:
32         if item[a_key_string] in classes_observed:
33             classes_observed[item[a_key_string]] += 1
34         else:
35             classes_observed[item[a_key_string]] = 1
36     return(classes_observed)
37
38 def main():
39     with open('Meteorite_Landings.json', 'r') as f:
40         ml_data = json.load(f)
41
42     print(compute_average_mass(ml_data['meteorite_landings'], 'mass (g)'))
43
44     for row in ml_data['meteorite_landings']:
45         print(check_hemisphere(float(row['reclat']), float(row['reclong'])))
46
47     print(count_classes(ml_data['meteorite_landings'], 'recclass'))
48
49 if __name__ == '__main__':
50     main()
```

3.3.4 Additional Resources

- [Python3 Logging How To Guide](#)

3.4 Unit Testing

As our code continues to grow, how can we be sure it is working as expected? If we make minor changes to the code, what tests can we run to make sure we didn't break anything? Are our functions written well enough to capture and correctly handle all of the edge cases we throw at them? In this module, we will use the Python `pytest` library to write unit tests: small tests that are designed to test specific individual components of code. After working through this module, students should be able to:

- Find the documentation for the Python `pytest` library
- Identify parts of code that should be tested and appropriate assert methods
- Write and run reasonable unit tests

3.4.1 Getting Started

Unit tests are designed to test small components (e.g. individual functions) of your code. They should demonstrate that things that are expected to work actually do work, and things that are expected to break raise appropriate errors. The Python `pytest` unit testing framework supports test automation, set up and shut down code for tests, and aggregation of tests into collections. It is not part of the Python Standard Library, so we must install it.

```
[isp02]$ pip3 install --user pytest
```

Find the [documentation](#) here.

Pull a copy of the [meteorite landings data analysis script](#) we have been working on, and a copy of the [meteorite landings json file](#), if you don't have copies already.

3.4.2 Devise a Reasonable Test

The functions in this Python3 script are relatively simple, but how can we be sure they are working as intended? Let's begin with the `compute_average_mass()` function. We might choose to test it manually using the Python3 interactive interpreter:

```
>>> from ml_data_analysis import compute_average_mass
>>>
>>> data = [{'thing': 1}, {'thing': 2}]
>>>
>>> print(compute_average_mass(data, 'thing'))
1.5
```

So simple! We import our code, hand-craft a simple data structure, and send the data plus the key we are interested in to our function. We know off the top of our heads that the average of 1 and 2 is 1.5, and that is in fact the number we get back.

Instead of writing that out each time we want to test, let's instead put this into another Python3 script. When writing test scripts, it is a common convention to name them the same name as the script you are testing, but with the `test_` prefix added at the beginning.

```
[isp02]$ ls
Meteorite_Landings.json  ml_data_analysis.py
[isp02]$ touch test_ml_data_analysis.py
[isp02]$ ls
Meteorite_Landings.json  ml_data_analysis.py  test_ml_data_analysis.py
```

Open up the script with VIM and put in our testing code from before:

```
1 from ml_data_analysis import compute_average_mass
2
3 data = [{'thing': 1}, {'thing': 2}]
4 print(compute_average_mass(data, 'thing'))
```

Next try to execute the test script on the command line:

```
[isp02]$ python3 test_ml_data_analysis.py
1.5
```

Great! We assume the test is working. But we still have to look at the output (1.5) and remember back to our hand-crafted data and make sure that is the correct result. It would be more efficient if we had a way to check that the correct answer is returned in our test script itself. To do this, we can use the `assert` statement.

```
1 from ml_data_analysis import compute_average_mass
2
3 data = [{'thing': 1}, {'thing': 2}]
4 assert(compute_average_mass(data, 'thing') == 1.5)
```

Now instead of printing the result, we use `assert` to make sure it is equal to our expected outcome. If the conditional is true, nothing will be printed. If the conditional is false, we will see an `AssertionError`.

EXERCISE

- Write a few more tests to convince yourself that the function is in fact returning the average of the input values.
- Modify one of the tests so that it should fail, and execute the tests to confirm that it does fail.
- If you have multiple tests that pass and multiple tests that fail, how would you know?

3.4.3 Automate Testing with Pytest

Pytest is an excellent framework for small unit tests and for large functional tests (as we will see later in the semester). If you previously installed `pytest` with `pip3`, now would be a good time to double check that the installation worked and there is an executable called `pytest` in your `PATH`:

```
[isp02]$ pytest --version
pytest 7.0.0
```

Next, we just need to make a minor organizational change to our test code. Pytest will automatically look in our working tree for files that start with the `test_` prefix, and execute the tests within.

```
1 from ml_data_analysis import compute_average_mass
2
3 def test_compute_average_mass():
```

(continues on next page)

(continued from previous page)

```

4  assert compute_average_mass([{'a': 1}, {'a': 2}], 'a') == 1.5
5  assert compute_average_mass([{'a': 1}, {'a': 2}, {'a': 3}], 'a') == 2
6  assert compute_average_mass([{'a': 10}, {'a': 1}, {'a': 1}], 'a') == 4

```

Call the `pytest` executable in your top directory, it will find your test function in your test script, run that function, and finally print some informative output:

```

===== test session starts
platform linux -- Python 3.6.8, pytest-7.0.0, pluggy-1.0.0
rootdir: /home/wallen/coe-332/code-organization
collected 1 item

test_ml_data_analysis.py .
[100%]

===== 1 passed in 0.01s

```

3.4.4 What Else Should We Test?

The simple tests we wrote above seem almost trivial, but they are actually great sanity tests to tell us that our code is working. What other behaviors of our `compute_average_mass()` function should we test? In no particular order, we could test the following non-exhaustive list:

- If the list only contains one dictionary object, the function still behaves as expected
- The return value should be type `float`
- If we send it an empty list, that should raise some sort of exception
- If we send it a list of non-uniform dictionaries (e.g. the dictionaries don't all have the expected key), we should get a `KeyError`
- If we send it bad values (e.g. a value is a string instead of an expected float), we should get a `ValueError`
- If we send it a string that doesn't appear in the dictionaries, we should get a `KeyError`

Tip: A list of all of the built-in Python3 exceptions can be found in the [Python docs](#).

To test some of these behaviors, let's create some additional assertions and organize them into their own functions.

```

1  from ml_data_analysis import compute_average_mass
2  import pytest
3
4  def test_compute_average_mass():
5      assert compute_average_mass([{'a': 1}], 'a') == 1
6      assert compute_average_mass([{'a': 1}, {'a': 2}], 'a') == 1.5
7      assert compute_average_mass([{'a': 1}, {'a': 2}, {'a': 3}], 'a') == 2
8      assert compute_average_mass([{'a': 10}, {'a': 1}, {'a': 1}], 'a') == 4
9      assert isinstance(compute_average_mass([{'a': 1}, {'a': 2}], 'a'), float) == True
10
11 def test_compute_average_mass_exceptions():

```

(continues on next page)

(continued from previous page)

```

12     with pytest.raises(ZeroDivisionError):
13         compute_average_mass([], 'a')           # send an empty list
14     with pytest.raises(KeyError):
15         compute_average_mass([{'a': 1}, {'b': 1}], 'a')   # dictionaries not
↳ uniform
16     with pytest.raises(ValueError):
17         compute_average_mass([{'a': 1}, {'a': 'x'}], 'a') # value not a float
18     with pytest.raises(KeyError):
19         compute_average_mass([{'a': 1}, {'a': 2}], 'b')   # key not in dicts

```

After adding the above tests, run pytest again:

```

===== test session starts
↳ =====
platform linux -- Python 3.6.8, pytest-7.0.0, pluggy-1.0.0
rootdir: /home/wallen/coe-332/code-organization
collected 2 items

test_ml_data_analysis.py ..
↳ [100%]

===== 2 passed in 0.01s
↳ =====

```

Success! The tests for our first function are passing. Our test suite essentially documents our intent for the behavior of the `compute_average_mass()` function. And, if ever we change the code in that function, we can see if the behavior we intend still passes the test.

EXERCISE

In the same test script, but under new test function definitions:

- Write tests for the `check_hemisphere()` function
- Write tests for the `count_classes()` function

3.4.5 Additional Resources

- [Pytest documentation.](#)
- [Exceptions in Python](#)

UNIT 4: CONTAINERIZATION AND AUTOMATION

In Unit 4 we begin to isolate our work from our development environment through a process called *containerization*. We will learn how to use existing containers, build our own containers, and push containers to a public container registry. We will also see how to automate some of our more tedious processes in order to speed up the development cycle.

4.1 Introduction to Containers

Containers are an important common currency for app development, web services, scientific computing, and more. Containers allow you to package an application along with all of its dependencies, isolate it from other applications and services, and deploy it consistently and reproducibly and *platform-agnostically*. In this introductory module, we will learn about containers and their uses, in particular the containerization platform **Docker**.

After going through this module, students should be able to:

- Describe what a container is
- Use essential docker commands
- Find and pull existing containers from Docker Hub
- Run containers interactively and non-interactively

4.1.1 What is a Container?

- A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another.
- Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package.
- Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space, hence are *lightweight* and have *low overhead*.
- Containers ensure *portability* and *reproducibility* by isolating the application from environment.

4.1.2 How is a Container Different from a VM?

Virtual machines enable application and resource isolation, run on top of a hypervisor (high overhead). Multiple VMs can run on the same physical infrastructure - from a few to dozens depending on resources. VMs take up more disk space and have long start up times (~minutes).

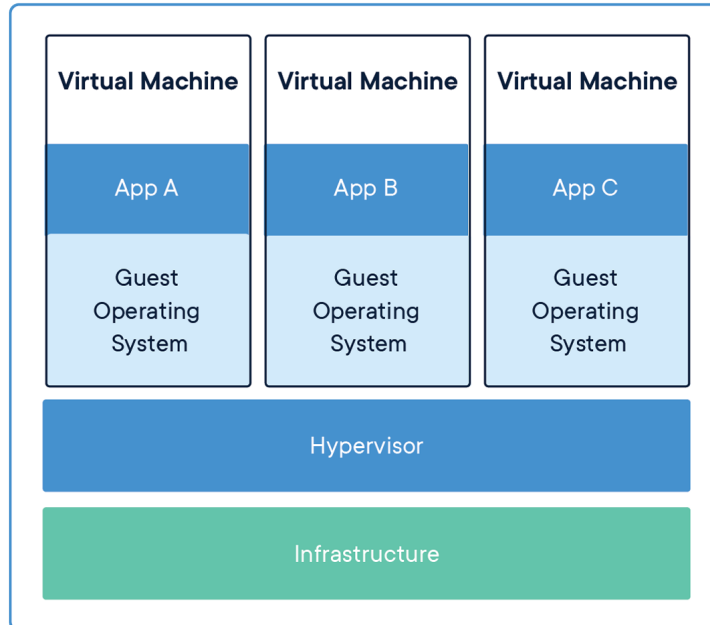


Fig. 1: Applications isolated by VMs.

Containers enable application and resource isolation, run on top of the host operating system. Many containers can run on the same physical infrastructure - up to 1,000s depending on resources. Containers take up less disk space than VMs and have very short start up times (~100s of ms).

4.1.3 Docker

Docker is a containerization platform that uses OS-level virtualization to package software and dependencies in deliverable units called containers. It is by far the most common containerization platform today, and most other container platforms are compatible with Docker. (E.g. Singularity and Shifter are two containerization platforms you'll find in HPC environments).

We can find existing containers at:

1. [Docker Hub](#)
2. [Quay.io](#)
3. [BioContainers](#)

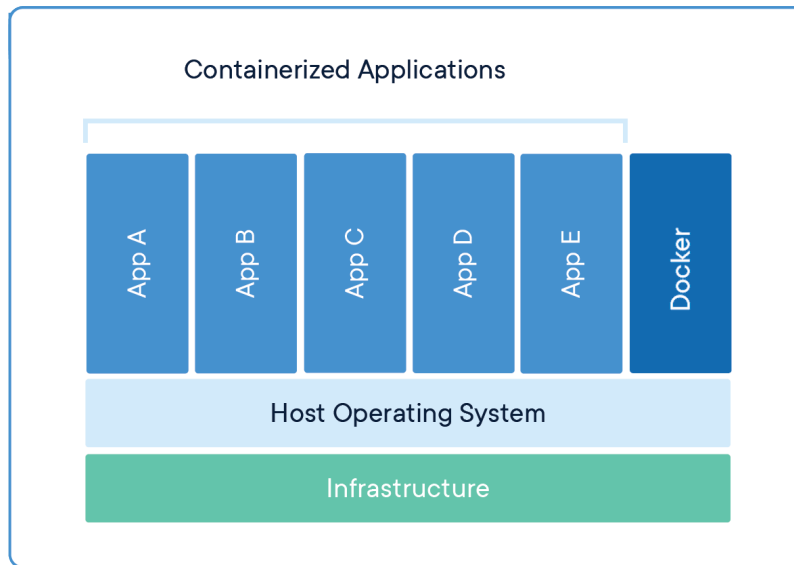


Fig. 2: Applications isolated by containers.

4.1.4 Some Quick Definitions

Container

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. Containers includes everything from the operating system, user-added files, metadata.

Image

A Docker images is a read-only file used to produce Docker containers. It is comprised of layers of other images, and any changes made to an image can only be saved and propagated on by adding new layers. The “base image” is the bottom-most layer that does not depend on any other layer and typically defines, e.g., the operating system for the container. Running a Docker image creates an instance of a Docker container.

Dockerfile

The Dockerfile is a recipe for creating a Docker image. They are simple, usually short plain text files that contain a sequential set of commands (*a recipe*) for installing and configuring your application and all of its dependencies. The Docker command line interface is used to “build” an image from a Dockerfile.

Image Registry

The Docker images you build can be stored in online image registries, such as [Docker Hub](#). (It is similar to the way we store Git repositories on GitHub.) Image registries support the notion of tags on images to identify specific versions of images. It is mostly public, and many “official” images can be found.

4.1.5 Summing Up

If you are developing an app or web service, you will almost certainly want to work with containers. First you must either **build** an image from a Dockerfile, or **pull** an image from a public registry. Then, you **run** (or deploy) an instance of your image into a container. The container represents your app or web service, running in the wild, isolated from other apps and services.

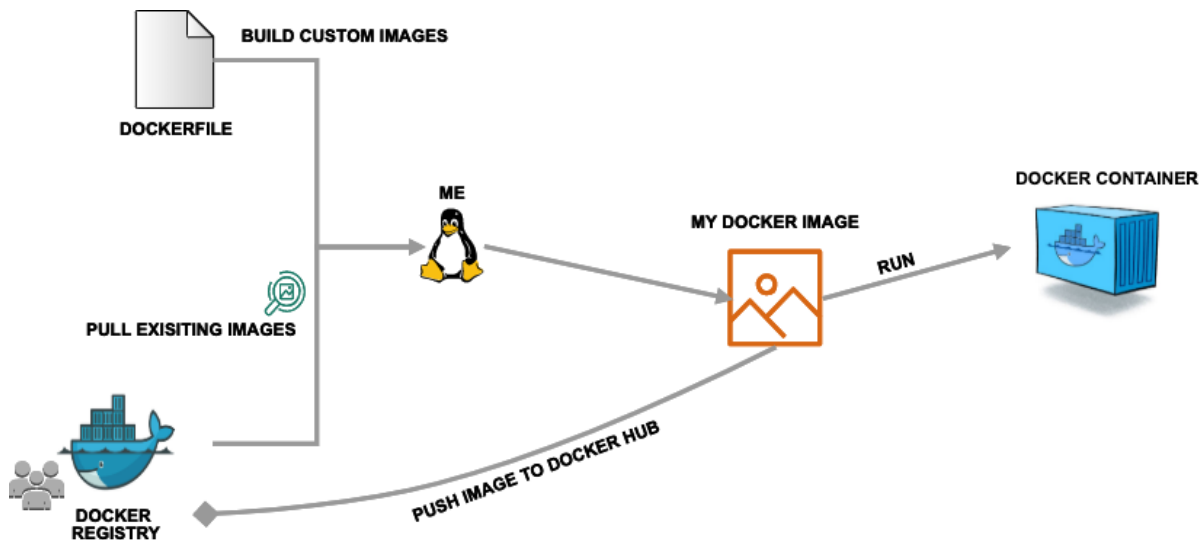


Fig. 3: Simple Docker workflow.

4.1.6 Getting Started With Docker

Much like the `git` command line tools, the `docker` command line tools follow the syntax: `docker <verb> <parameters>`. Discover all the verbs available by typing `docker --help`, and discover help for each verb by typing `docker <verb> --help`. Open up your favorite terminal, log in to the class server, and try running the following:

```
[isp02]$ docker version
Client: Docker Engine - Community
Version:      20.10.12
API version:  1.41
Go version:   go1.16.12
Git commit:   e91ed57
```

(continues on next page)

(continued from previous page)

```

Built:           Mon Dec 13 11:45:41 2021
OS/Arch:         linux/amd64
Context:         default
Experimental:    true

Server: Docker Engine - Community
Engine:
  Version:       20.10.12
  API version:   1.41 (minimum version 1.12)
  Go version:    go1.16.12
  Git commit:    459d0df
  Built:        Mon Dec 13 11:44:05 2021
  OS/Arch:      linux/amd64
  Experimental:  false
containerd:
  Version:      1.4.12
  GitCommit:    7b11cfaabd73bb80907dd23182b9347b4245eb5d
runc:
  Version:      1.0.2
  GitCommit:    v1.0.2-0-g52b36a2
docker-init:
  Version:      0.19.0
  GitCommit:    de40ad0

```

Warning: Please let the instructors know if you get any errors on issuing the above command.

EXERCISE

Take a few minutes to run `docker --help` and a few examples of `docker <verb> --help` to make sure you can find and read the help text.

4.1.7 Working with Images from Docker Hub

To introduce ourselves to some of the most essential Docker commands, we will go through the process of listing images that are currently available on the ISP server, we will pull a ‘hello-world’ image from Docker Hub, then we will run the ‘hello-world’ image to see what it says.

List images on the ISP server with the `docker images` command. This peaks into the Docker daemon, which is shared by all users on this system, to see which images are available, when they were created, and how large they are:

```

[isp02]$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
eriksfi/pi-estimator 0.1-mpi            7c0c51eadb4a       5 days ago         2.4GB
ubuntu              18.04              886eca19e611       5 weeks ago        63.1MB
eriksfi/ucvm_1210_cvms5 11162039           2e60f37e22b0       2 months ago       6.35GB

```

(continues on next page)

(continued from previous page)

eriksf/bert-classifier	0.0.1	f0f041a7aa2f	4 months ago	↗
↪ 6.67GB				
eriksf/bert-classifier	0.0.2	f0f041a7aa2f	4 months ago	↗
↪ 6.67GB				
tacc/tacc-ml	ubuntu16.04-cuda10-tf2.4-pt1.7	2dfb4d60a1ee	4 months ago	↗
↪ 6.06GB				
tacc/tacc-ml	centos7-cuda10-tf2.4-pt1.7	e150e2a64b12	4 months ago	↗
↪ 6.34GB				
tacc/tacc-ml	centos7-cuda10-tf2.1-pt1.3	152aac5c5d4a	4 months ago	↗
↪ 6.51GB				
tacc/tacc-ml	centos7-cuda10-tf1.15-pt1.3	59d9b476cda9	4 months ago	↗
↪ 6.74GB				
tacc/tacc-ml	centos7-cuda9-tf1.14-pt1.3	4c5b5c69912b	4 months ago	↗
↪ 6.38GB				
tacc/tacc-ml	ubuntu16.04-cuda10-tf2.1-pt1.3	11749adce91f	4 months ago	↗
↪ 6.23GB				
tacc/tacc-ml	ubuntu16.04-cuda10-tf1.15-pt1.3	53408ed25cc8	4 months ago	↗
↪ 6.46GB				
tacc/tacc-ml	ubuntu16.04-cuda9-tf1.14-pt1.3	1752a789aa75	4 months ago	↗
↪ 6.1GB				
tacc/tacc-ml	ubuntu16.04	952f3930bbfc	4 months ago	↗
↪ 543MB				
tacc/tacc-ml	centos7	068a63ee19c8	4 months ago	↗
↪ 814MB				
centos	7	eeb6ee3f44bd	5 months ago	↗
↪ 204MB				
centos	8	5d0da3dc9764	5 months ago	↗
↪ 231MB				

Pull an image from Docker hub with the `docker pull` command. This looks through the Docker Hub registry and downloads the 'latest' version of that image:

```
[isp02]$ docker pull hello-world
Using default tag: latest
latest: Pulling from library/hello-world
0e03bdcc26d7: Pull complete
Digest: sha256:31b9c7d48790f0d8c50ab433d9c3b7e17666d6993084c002c2ff1ca09b96391d
Status: Downloaded newer image for hello-world:latest
docker.io/library/hello-world:latest
```

Run the image we just pulled with the `docker run` command. In this case, running the container will execute a simple shell script inside the container that has been configured as the 'default command' when the image was built:

```
[isp02]$ docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
```

(continues on next page)

(continued from previous page)

3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

To try something more ambitious, you can run an Ubuntu container with:

```
$ docker run -it ubuntu bash
```

Share images, automate workflows, and more with a free Docker ID:

<https://hub.docker.com/>

For more examples and ideas, visit:

<https://docs.docker.com/get-started/>

Check to see if any containers are still running using `docker ps`:

```
[isp02]$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
--------------	-------	---------	---------	--------	-------	-------

EXERCISE

The command `docker ps` shows only currently running containers. Pull up the help text for that command and figure out how to show all containers, not just currently running containers.

4.1.8 Pull Another Image

Navigate to the repositories of user `wjallen` on Docker Hub [here](#).

Scroll down to find an image called `wjallen/bsd`, then click on that image.

Pull the image using the suggested command, then check to make sure it is available locally:

```
[isp02]$ docker pull wjallen/bsd:1.0
...
[isp02]$ docker images
...
[isp02]$ docker inspect wjallen/bsd:1.0
...
```

Tip: Use `docker inspect` to find some metadata available for each image.

4.1.9 Start an Interactive Shell Inside a Container

Using an interactive shell is a great way to poke around inside a container and see what is in there. Imagine you are ssh-ing to a different Linux server, have root access, and can see what files, commands, environment, etc., is available.

Before starting an interactive shell inside the container, execute the following commands on the ISP server (we will see why in a minute):

```

[isp02]$ whoami
wallen
[isp02]$ pwd
/home/wallen
[isp02]$ cat /etc/os-release
NAME="CentOS Linux"
VERSION="7 (Core)"
ID="centos"
ID_LIKE="rhel fedora"
VERSION_ID="7"
PRETTY_NAME="CentOS Linux 7 (Core)"
ANSI_COLOR="0;31"
CPE_NAME="cpe:/o:centos:centos:7"
HOME_URL="https://www.centos.org/"
BUG_REPORT_URL="https://bugs.centos.org/"

CENTOS_MANTISBT_PROJECT="CentOS-7"
CENTOS_MANTISBT_PROJECT_VERSION="7"
REDHAT_SUPPORT_PRODUCT="centos"
REDHAT_SUPPORT_PRODUCT_VERSION="7"

[isp02]$ ls -l /usr/games/
total 0

```

Now start the interactive shell:

```

[isp02]$ docker run --rm -it wjallen/bsd:1.0 /bin/bash
root@fc5b620c5a88:/#

```

Here is an explanation of the command options:

docker run	# run a container
--rm	# remove the container when we exit
-it	# interactively attach terminal to inside of container
wjallen/bsd:1.0	# image and tag on local machine
/bin/bash	# shell to start inside container

Try the following commands - the same commands you did above before starting the interactive shell in the container - and note what has changed:

```

root@fc5b620c5a88:/# whoami
root
root@fc5b620c5a88:/# pwd
/
root@fc5b620c5a88:/# cat /etc/os-release
NAME="Ubuntu"
VERSION="16.04.6 LTS (Xenial Xerus)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 16.04.6 LTS"
VERSION_ID="16.04"
HOME_URL="http://www.ubuntu.com/"
SUPPORT_URL="http://help.ubuntu.com/"

```

(continues on next page)

(continued from previous page)

```

BUG_REPORT_URL="http://bugs.launchpad.net/ubuntu/"
VERSION_CODENAME=xenial
UBUNTU_CODENAME=xenial
root@fc5b620c5a88:/# ls /usr/games/
adventure  bcd      countmail  hack    morse    ppt      robots  teachgammon  worms
arithmetic boggle   cribbage   hangman number   primes   rot13    tetris-bsd   wtf
atc        caesar   dab        hunt    phantasia  quiz    sail     trek         wump
backgammon canfield go-fish    mille   pig       rain     snake    wargames
battlestar cfscores gomoku     monop   pom       random   snscore  worm

```

Now you are the `root` user on a different operating system inside a running Linux container! You can type `exit` to escape the container.

EXERCISE

Before you exit the container, try running a few of the games (e.g. `hangman`).

4.1.10 Run a Command Inside a Container

Back out on the ISP server, we now know we have an image called `wjallen/bsd:1.0` that has some terminal games inside it which would not otherwise be available to us on the ISP server. They (and their dependencies) are *isolated* from everything else. This image (`wjallen/bsd:1.0`) is portable and will run the exact same way on any OS that Docker supports.

In practice, though, we don't want to start interactive shells each time we need to use a software application inside an image. Docker allows you to spin up an *ad hoc* container to run applications from outside. For example, try:

```

[isp02]$ docker run --rm wjallen/bsd:1.0 whoami
root
[isp02]$ docker run --rm wjallen/bsd:1.0 pwd
/
[isp02]$ docker run --rm wjallen/bsd:1.0 cat /etc/os-release
NAME="Ubuntu"
VERSION="16.04.6 LTS (Xenial Xerus)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 16.04.6 LTS"
VERSION_ID="16.04"
HOME_URL="http://www.ubuntu.com/"
SUPPORT_URL="http://help.ubuntu.com/"
BUG_REPORT_URL="http://bugs.launchpad.net/ubuntu/"
VERSION_CODENAME=xenial
UBUNTU_CODENAME=xenial
[isp02]$ docker run --rm wjallen/bsd:1.0 ls /usr/games
adventure  bcd      countmail  hack    morse    ppt      robots  teachgammon  worms
arithmetic boggle   cribbage   hangman number   primes   rot13    tetris-bsd   wtf
atc        caesar   dab        hunt    phantasia  quiz    sail     trek         wump
backgammon canfield go-fish    mille   pig       rain     snake    wargames
battlestar cfscores gomoku     monop   pom       random   snscore  worm
[isp02]$ docker run --rm -it wjallen/bsd:1.0 hangman

```

The first four commands above omitted the `-it` flags because they did not require an interactive terminal to run. On each of these commands, Docker finds the image the command refers to, spins up a new container based on that image, executes the given command inside, prints the result, and exits and removes the container.

The last command, which executes the hangman game, requires an interactive terminal so the `-it` flags were provided.

4.1.11 Essential Docker Command Summary

Command	Usage
<code>docker login</code>	Authenticate to Docker Hub using username and password
<code>docker images</code>	List images on the local machine
<code>docker ps</code>	List containers on the local machine
<code>docker pull</code>	Download an image from Docker Hub
<code>docker run</code>	Run an instance of an image (a container)
<code>docker inspect</code>	Provide detailed information on Docker objects
<code>docker rmi</code>	Delete an image
<code>docker rm</code>	Delete a container
<code>docker stop</code>	Stop a container
<code>docker build</code>	Build a docker image from a Dockerfile in the current working directory
<code>docker tag</code>	Add a new tag to an image
<code>docker push</code>	Upload an image to Docker Hub

If all else fails, display the help text:

```
[isp02]$ docker --help
shows all docker options and summaries
```

```
[isp02]$ docker COMMAND --help
shows options and summaries for a particular command
```

4.1.12 Additional Resources

- [Docker Docs](#)
- [Best practices for writing Dockerfiles](#)
- [Docker Hub](#)
- [Docker for Beginners](#)
- [Play with Docker](#)

4.2 Advanced Containers

In the first part, we pulled and ran existing container images from Docker Hub. In this section, we will build an image from scratch for running some of our own Python3 code. Then, we will push that image back up to Docker Hub so others may find and use it. After going through this module, students should be able to:

- Install and test code in a container interactively
- Write a Dockerfile from scratch
- Build a Docker image from a Dockerfile

- Push a Docker image to Docker Hub

4.2.1 Getting Set Up

Scenario: You are a developer who has written some code for reading and parsing meteorite landing data in JSON format. You now want to distribute that code for others to use in what you know to be a stable production environment (including OS and dependency versions). End users may want to use this application on their local workstations, in the cloud, or on an HPC cluster.

The first step in a typical container development workflow entails installing and testing an application interactively within a running Docker container.

Note: We recommend doing this on the class ISP server. But, one of the most important features of Docker is that it is platform agnostic. These steps could be done anywhere Docker is installed.

To begin, make a new folder for this work and prepare to gather some important files.

```
[isp02]$ cd ~/coe-332/
[isp02]$ mkdir docker-exercise/
[isp02]$ cd docker-exercise/
[isp02]$ pwd
/home/wallen/coe-332/docker-exercise
```

Specifically, you need your `ml_data_analysis.py` script and the input data file called `Meteorite_Landings.json`. You can make copies of your own, or download sample copies from the links below. You also need a `Dockerfile`, and we can just make an empty one with no contents for now.

```
[isp02]$ pwd
/home/wallen/coe-332/docker-exercise
[isp02]$ touch Dockerfile
[isp02]$ wget https://raw.githubusercontent.com/tacc/coe-332-sp22/main/docs/unit04/
↳scripts/Meteorite_Landings.json
[isp02]$ wget https://raw.githubusercontent.com/tacc/coe-332-sp22/main/docs/unit04/
↳scripts/ml_data_analysis.py
[isp02]$ ls
Dockerfile  Meteorite_Landings.json  ml_data_analysis.py
```

Warning: It is important to carefully consider what files and folders are in the same PATH as a Dockerfile (known as the ‘build context’). The `docker build` process will index and send all files and folders in the same directory as the Dockerfile to the Docker daemon, so take care not to `docker build` at a root level.

4.2.2 Containerize Code Interactively

There are several questions you must ask yourself when preparing to containerize code for the first time:

1. What is an appropriate base image?
2. What dependencies are required for my program?
3. What is the install process for my program?
4. What environment variables may be important?

We can work through these questions by performing an **interactive installation** of our Python script. Our development environment (the class ISP server) is a Linux server running CentOS 7.7. We know our code works here, so that is how we will containerize it. Use `docker run` to interactively attach to a fresh [CentOS 7.7 container](#).

Warning: Due to Log4Shell vulnerability (CVE-2021-44228 or CVE-2021-45046), let's instead pull CentOS 7.9.

```
[isp02]$ docker run --rm -it -v $PWD:/code centos:7.9.2009 /bin/bash
[root@7ad568453e0b /]#
```

Here is an explanation of the options:

<code>docker run</code>	<code># run a container</code>
<code>--rm</code>	<code># remove the container on exit</code>
<code>-it</code>	<code># interactively attach terminal to inside of container</code>
<code>-v \$PWD:/code</code>	<code># mount the current directory to /code</code>
<code>centos:7.9.2009</code>	<code># image and tag from Docker Hub</code>
<code>/bin/bash</code>	<code># shell to start inside container</code>

The command prompt will change, signaling you are now 'inside' the container. And, new to this example, we are using the `-v` flag which mounts the contents of our current directory (`$PWD`) inside the container in a folder in the root directory called (`/code`).

Update and Upgrade

The first thing we will typically do is use the CentOS package manager `yum` to update the list of available packages and install newer versions of the packages we have. We can do this with:

```
[root@7ad568453e0b /]# yum update
...
```

Note: You will need to press 'y' followed by 'Enter' twice to download and install the updates

Install Required Packages

For our Python scripts to work, we need to install two dependencies: Python3 and the ‘pytest’ package (more on the ‘pytest’ package later, let’s just assume for now we need it).

```
[root@7ad568453e0b /]# yum install python3
...
[root@7ad568453e0b /]# python3 --version
Python 3.6.8
[root@7ad568453e0b /]# pip3 install pytest==7.0.0
Collecting pytest==7.0.0
...
Installing collected packages: py, pyparsing, packaging, typing-extensions, zipp,
  importlib-metadata, pluggy, attrs, iniconfig, tomli, pytest
Successfully installed attrs-21.4.0 importlib-metadata-4.8.3 iniconfig-1.1.1 packaging-
  21.3
  pluggy-1.0.0 py-1.11.0 pyparsing-3.0.7 pytest-7.0.0 tomli-1.2.3 typing-extensions-4.1.1
  zipp-3.6.0
```

Warning: An important question to ask is: Does the versions of Python and other dependencies match the versions you are developing with in your local environment? If not, make sure to install the correct version of Python.

Install and Test Your Code

At this time, we should make a small edit to the code that will make it a little more flexible and more amenable to running in a container. Instead of hard coding the filename ‘Meteorite_Landings.json’ in the script, let’s make a slight modification so we can pass the filename on the command line. In the script, add this line near the top:

```
import sys
```

And change the with open... statements to these, as appropriate:

```
with open(sys.argv[1], 'r') as f:
    ml_data = json.load(f)
```

Since we are using a simple Python script, there is not a difficult install process. However, we can make it executable and add it to them user’s *PATH*.

```
[root@7ad568453e0b /]# cd /code
[root@7ad568453e0b /]# chmod +rx ml_data_analysis.py
[root@7ad568453e0b /]# export PATH=/code:$PATH
```

Now test with the following:

```
[root@7ad568453e0b /]# cd /home
[root@7ad568453e0b /]# cp /code/Meteorite_Landings.json .
[root@7ad568453e0b /]# ml_data_analysis.py Meteorite_Landings.json
83857.3
Northern & Eastern
...etc
```

We now have functional versions of our script ‘installed’ in this container. Now would be a good time to execute the *history* command to see a record of the build process. When you are ready, type *exit* to exit the container and we can start writing these build steps into a Dockerfile.

4.2.3 Assemble a Dockerfile

After going through the build process interactively, we can translate our build steps into a Dockerfile using the directives described below. Open up your copy of `Dockerfile` with a text editor and enter the following:

The FROM Instruction

We can use the FROM instruction to start our new image from a known base image. This should be the first line of our Dockerfile. In our scenario, we want to match our development environment with CentOS 7.9. We know our code works in that environment, so that is how we will containerize it for others to use:

```
FROM centos:7.9.2009
```

Base images typically take the form *os:version*. Avoid using the ‘latest’ version; it is hard to track where it came from and the identity of ‘latest’ can change.

Tip: Browse [Docker Hub](#) to discover other potentially useful base images. Keep an eye out for the ‘Official Image’ badge.

The RUN Instruction

We can install updates, install new software, or download code to our image by running commands with the RUN instruction. In our case, our only dependencies were Python3 and the “pytest” library. So, we will use a few RUN instructions to install them. Keep in mind that the the `docker build` process cannot handle interactive prompts, so we use the `-y` flag with `yum` and `pip3`.

```
RUN yum update -y
RUN yum install -y python3
RUN pip3 install pytest==7.0.0
```

Each RUN instruction creates an intermediate image (called a ‘layer’). Too many layers makes the Docker image less performant, and makes building less efficient. We can minimize the number of layers by combining RUN instructions. Dependencies that are more likely to change over time (e.g. Python3 libraries) still might be better off in their own RUN instruction in order to save time building later on:

```
RUN yum update -y && \
    yum install -y python3

RUN pip3 install pytest==7.0.0
```

Tip: In the above code block, the character at the end of the lines causes the newline character to be ignored. This can make very long run-on lines with many commands separated by `&&` easier to read.

The COPY Instruction

There are a couple different ways to get your source code inside the image. One way is to use a RUN instruction with `wget` to pull your code from the web. When you are developing, however, it is usually more practical to copy code in from the Docker build context using the COPY instruction. For example, we can copy our script to the root-level `/code` directory with the following instructions:

```
COPY ml_data_analysis.py /code/ml_data_analysis.py
```

And, don't forget to perform another RUN instruction to make the script executable:

```
RUN chmod +rx /code/ml_data_analysis.py
```

The ENV Instruction

Another useful instruction is the ENV instruction. This allows the image developer to set environment variables inside the container runtime. In our interactive build, we added the `/code` folder to the `PATH`. We can do this with ENV instructions as follows:

```
ENV PATH "/code:$PATH"
```

Putting It All Together

The contents of the final Dockerfile should look like:

```
1 FROM centos:7.9.2009
2
3 RUN yum update -y && \
4     yum install -y python3
5
6 RUN pip3 install pytest==7.0.0
7
8 COPY ml_data_analysis.py /code/ml_data_analysis.py
9
10 RUN chmod +rx /code/ml_data_analysis.py
11
12 ENV PATH "/code:$PATH"
```

4.2.4 Build the Image

Once the Dockerfile is written and we are satisfied that we have minimized the number of layers, the next step is to build an image. Building a Docker image generally takes the form:

```
[isp02]$ docker build -t <dockerhubusername>/<code>:<version> .
```

The `-t` flag is used to name or 'tag' the image with a descriptive name and version. Optionally, you can preface the tag with your **Docker Hub username**. Adding that namespace allows you to push your image to a public registry and share it with others. The trailing dot `.` in the line above simply indicates the location of the Dockerfile (a single `.` means 'the current directory').

To build the image, use:

```
[isp02]$ docker build -t username/ml_data_analysis:1.0 .
```

Note: Don't forget to replace 'username' with your Docker Hub username.

Use `docker images` to ensure you see a copy of your image has been built. You can also use `docker inspect` to find out more information about the image.

```
[isp02]$ docker images
REPOSITORY          TAG          IMAGE ID          CREATED           SIZE
wjallen/ml_data_analysis  1.0         2883079fad18     About a minute ago  547MB
...
```

```
[isp02]$ docker inspect username/ml_data_analysis:1.0
```

If you need to rename your image, you can either re-tag it with `docker tag`, or you can remove it with `docker rmi` and build it again. Issue each of the commands on an empty command line to find out usage information.

4.2.5 Test the Image

We can test a newly-built image two ways: interactively and non-interactively. In interactive testing, we will use `docker run` to start a shell inside the image, just like we did when we were building it interactively. The difference this time is that we are NOT mounting the code inside with the `-v` flag, because the code is already in the container:

```
[isp02]$ docker run --rm -it username/ml_data_analysis:1.0 /bin/bash
...
[root@c5cf05edddcd /]# ls /code
ml_data_analysis.py
[root@c5cf05edddcd /]# cd /home
[root@c5cf05edddcd home]# pwd
/home
[root@c5cf05edddcd home]# ml_data_analysis.py Meteorite_Landings.json
Traceback (most recent call last):
  File "/code/ml_data_analysis.py", line 96, in <module>
    main()
  File "/code/ml_data_analysis.py", line 82, in main
    with open(sys.argv[1], 'r') as f:
FileNotFoundError: [Errno 2] No such file or directory: 'Meteorite_Landings.json'
```

Here is an explanation of the options:

```
docker run      # run a container
--rm           # remove the container when we exit
-it            # interactively attach terminal to inside of container
username/...    # image and tag on local machine
/bin/bash      # shell to start inside container
```

Uh oh! We forgot about `Meteorite_Landings.json`! We get a `FileNotFoundError` in Python3. This is because we (1) did not copy the JSON file into the container at build time, and (2) we did not copy the JSON file into the container at run time.

We should pause at this moment to think about how we want to distribute this application. Should the data be encapsulated within? Or should we expect potential users to be bringing their own data for analysis?

Let's try again, but this time mount the data inside the container so we can access it. If we mount the current folder as, e.g., /data, then everything in the current folder will be available. In addition, if we write any new files inside the container to /data, those will be preserved and persist outside the container once it stops.

```
[isp02]$ docker run --rm -it -v $PWD:/data username/ml_data_analysis:1.0 /bin/bash
[root@dc0d6bf1875c /]# pwd
/
[root@dc0d6bf1875c /]# ls /data
Dockerfile  Meteorite_Landings.json  ml_data_analysis.py
[root@dc0d6bf1875c /]# ls /code
ml_data_analysis.py
[root@dc0d6bf1875c /]# ml_data_analysis.py /data/Meteorite_Landings.json
83857.3
Northern & Eastern
... etc
```

Everything looks like it works now! Next, exit the container and test the code non-interactively. Notice we are calling the container again with `docker run`, but instead of specifying an interactive (`-it`) run, we just issue the command as we want to call it on the command line. Also, notice the return of the `-v` flag, because we need to create a volume mount so that our data (`Meteorite_Landings.json`) is available inside the container.

```
[isp02]$ docker run --rm -v $PWD:/data username/ml_data_analysis:1.0 ml_data_analysis.py
83857.3
Northern & Eastern
... etc
```

Much simpler and cleaner! Our only local dependencies are the Docker runtime and some input data that we provide. Then we pull and run the image, mounting our data inside the container and executing the embedded Python3 script. Anyone with their own data could follow our same steps to replicate our work in their own environments.

4.2.6 Share Your Docker Image

Now that you have containerized, tested, and tagged your code in a Docker image, the next step is to disseminate it so others can use it.

Docker Hub is the *de facto* place to share an image you built. Remember, the image must be name-spaced with either your Docker Hub username or a Docker Hub organization where you have write privileges in order to push it:

```
[isp02]$ docker login
...
[isp02]$ docker push username/ml_data_analysis:1.0
```

You and others will now be able to pull a copy of your container with:

```
[isp02]$ docker pull username/ml_data_analysis:1.0
```

As a matter of best practice, it is highly recommended that you store your Dockerfiles somewhere safe. A great place to do this is alongside the code in, e.g., GitHub. GitHub also has integrations to automatically update your image in the public container registry every time you commit new code. (More on this later in the semester).

For example, see: [Publishing Docker Images](#)

4.2.7 Additional Resources

- [Docker for Beginners](#)
- [Play with Docker](#)

4.3 Makefile Automation

The development cycle for Docker images can necessitate some fairly complex and long commands. It is easy to accidentally inject typos or forget part of the command. To speed up development in Docker, we will learn some simple automation techniques using Makefiles. The concepts learned here can be applied to many other areas, not just Docker containers. After going through this module, students should be able to:

- Write a Makefile for automating repetitive commands
- Execute specific targets in a Makefile
- Inject variables into a Makefile

4.3.1 Development Cycle

We previously talked at great length about why it is a good idea to containerize an app / service that you develop. As you write code, containerize, and test, you will be repeating some common commands over and over again. The “development cycle” in this case may follow the form:

1. Edit some source code (e.g. add a new function to your Python script)
2. Delete any running container with the old source code
3. Re-build the image with `docker build`
4. Start up a new container with `docker run`
5. Test the new code / functionality
6. Repeat

This 6-step cycle is great for iterating containers as you develop code. However, the commands we type out as we develop are potential error sources - it is easy to accidentally skip a step or inject typos.

4.3.2 Makefile

Makefiles can be a useful automation tool for testing your services. Many code projects use Makefiles to help with the compile and install process (e.g. `make` && `make install`). Here, we will set up a Makefile to help with the 6-step cycle above. Using certain keywords (called “targets”) we will create shortcuts to cleaning up running containers, re-building docker images, running new containers, and pushing images to Docker Hub.

Targets are listed in a file called `Makefile` in this format:

```
target: prerequisite(s)
      recipe
```

Targets are short keywords, and recipes are shell commands. For example, a simple target might look like:

```
images:
  docker images | grep wjallen
```

Put this text in a file called **Makefile** in your current directory, and then you simply need to type:

```
[isp02]$ make images
```

And that will list all the docker containers with the username 'jwallen' either in the image name or the container name. Makefiles can be further abstracted with variables to make them a little bit more flexible. Consider the following Makefile:

```
NAME ?= wjallen

all: images ps

images:
    docker images | grep ${NAME}

ps:
    docker ps -a | grep ${NAME}
```

Here we have added a variable **NAME** at the top so we can easily customize the targets below. We have also added two new targets: **ps** which lists running containers, and **all** which does not contain any recipes, but does contain two prerequisites - the other two targets. So these two are equivalent:

```
# make all targets
[isp02]$ make all

# or make them one-by-one
[isp02]$ make images
[isp02]$ make ps

# Try this out:
[isp02]$ NAME="bsd" make all
```

EXERCISE

Write a Makefile that:

1. Builds your Docker image in your name space
2. Runs a test command or test script to confirm your image is working
3. Pushes your image to Docker Hub

SOLUTION

The following would work for user **wjallen**. Remember, if any step results in an error, subsequent steps will not execute:

```
NAME ?= wjallen

all: build run push

images:
    docker images | grep ${NAME}
```

(continues on next page)

(continued from previous page)

```

ps:
    docker ps -a | grep ${NAME}

build:
    docker build -t ${NAME}/ml_data_analysis:1.0 .

run:
    docker run --rm -v ${PWD}:/data ${NAME}/ml_data_analysis:1.0 ml_data_analysis.py /
    ↪data/Meteorite_Landings.json

push:
    docker push ${NAME}/ml_data_analysis:1.0

```

Typing `make` on the command line will execute the first target, `all`, which in turn calls the `build`, `run`, and `push` targets in that order.

4.4 Docker Compose

Up to this point, we have been looking at single-container applications - small units of code that are containerized, executed *ad hoc* to generate or read a JSON file, then exit on completion. But what if we want to do something more complex? For example, what if our goal is to orchestrate a multi-container application consisting of, e.g., a Flask app, a database, a message queue, an authentication service, and more.

Docker compose is tool for managing multi-container applications. A YAML file is used to define all of the application service, and a few simple commands can be used to spin up or tear down all of the services.

In this module, we will get a first look at Docker compose. Later in this course we will do a deeper dive into advanced container orchestration. After going through this module, students should be able to:

- Translate Docker run commands into YAML files for Docker compose
- Run commands inside *ad hoc* containers using Docker compose
- Manage small software systems composed of more than one script, and more than one container
- Copy data into and out of containers as needed

4.4.1 Another Script, Another Container

We have been working a lot with a script for reading in and analyzing a JSON file of meteorite landing data. Let's quickly write a new script to generate that data, then we will package it into its own container. Consider the following script for **generating** the Meteorite Landing data we have been working with:

```

1  #!/usr/bin/env python3
2  import json
3  import random
4  import sys
5  import names
6
7  NUM = 10
8  CLASSES = ['CI1', 'CR2-an', 'CV3', 'EH4', 'H4', 'H5', 'H6', 'L5', 'L6', 'LL3-6', 'LL5']
9
10 def main():

```

(continues on next page)

(continued from previous page)

```

11 data = {'meteorite_landings': [{ for _ in range (NUM)]}
12
13
14 for i in range(NUM):
15     rand_lat = '{:.4f}'.format(random.uniform(-90.0000, 90.0000))
16     rand_lon = '{:.4f}'.format(random.uniform(-90.0000, 90.0000))
17     data['meteorite_landings'][i]['name'] = names.get_first_name()
18     data['meteorite_landings'][i]['id'] = str(10000 + 1 + i)
19     data['meteorite_landings'][i]['recclass'] = random.choice(CLASSES)
20     data['meteorite_landings'][i]['mass (g)'] = str(random.randrange(1, 10000))
21     data['meteorite_landings'][i]['reclat'] = rand_lat
22     data['meteorite_landings'][i]['reclong'] = rand_lon
23     data['meteorite_landings'][i]['GeoLocation'] = f'({rand_lat}, {rand_lon})'
24
25 with open(sys.argv[1], 'w') as o:
26     json.dump(data, o, indent=2)
27     print(f'Data written to {sys.argv[1]}!')
28
29 if __name__ == '__main__':
30     main()

```

Copy that into a file called `gen_ml_data.py`, save it, make it executable, and test it. You'll find that this script requires a **command line argument**. Meaning we have to invoke it AND pass some information on the command line in order to get it to work. In this case, it is expecting the name of the output file.

```

# copy contents into file called ``gen_ml_data.py`` and save
[isp02]$ chmod +rx gen_ml_data.py
[isp02]$ ./gen_ml_data.py
Traceback (most recent call last):
  File "./gen_ml_data.py", line 29, in <module>
    main()
  File "./gen_ml_data.py", line 25, in main
    with open(sys.argv[1], 'w') as o:
IndexError: list index out of range

[isp02]$ ./gen_ml_data.py data.json
Data written to data.json!
[isp02]$ ls
data.json  Dockerfile  gen_ml_data.py  Meteorite_Landings.json  ml_data_analysis.py
[isp02]$ head -n11 data.json
{
  "meteorite_landings": [
    {
      "name": "Sandra",
      "id": "10001",
      "recclass": "EH4",
      "mass (g)": "4119",
      "reclat": "73.8716",
      "reclong": "14.8207",
      "GeoLocation": "(73.8716, 14.8207)"
    },

```

Containerizing this script should be easy enough - we already worked through containerizing another very similar script.

Let's say for this new script we do not need the `pytest` dependency, because there is not really anything interesting to test. But, we do need a different dependency: the Python3 `names` library.

To make things a little more clear, rename the existing `Dockerfile` as `Dockerfile-analysis`, and make a copy of it called `Dockerfile-gen`.

```
[isp02]$ mv Dockerfile Dockerfile-analysis
[isp02]$ cp Dockerfile-analysis Dockerfile-gen
[isp02]$ ls
data.json          Dockerfile-analysis  Dockerfile-gen
gen_ml_data.py     Meteorite_Landings.json  ml_data_analysis.py
```

Edit `Dockerfile-gen` as follows:

```
1 FROM centos:7.9.2009
2
3 RUN yum update -y && \
4     yum install -y python3
5
6 RUN pip3 install names==0.3.0
7
8 COPY gen_ml_data.py /code/gen_ml_data.py
9
10 RUN chmod +rx /code/gen_ml_data.py
11
12 ENV PATH "/code:$PATH"
```

Now that we have a `Dockerfile` named something other than the default name, we need to modify our command line a little bit to build it:

```
[isp02]$ docker build -t username/gen_ml_data:1.0 -f Dockerfile-gen .
```

After the image is successfully built, change directories to a new folder just to be sure you are not running the local scripts or looking at the local data. Then, test the container as follows:

```
[isp02]$ mkdir test
[isp02]$ cd test
[isp02]$ ls
[isp02]$ docker run --rm username/gen_ml_data:1.0 gen_ml_data.py ml.json
Data written to ml.json!
```

If you list your local files, can you find `ml.json`? No! This is because whatever data generated inside the container is lost when the container completes its task. What we need to do is use the `-v` flag to mount a directory somewhere inside the container, write data into that directory, then the data will be captured after the container exists. For example:

```
[isp02]$ docker run --rm -v $PWD:/data username/gen_ml_data:1.0 gen_ml_data.py /data/ml.
↪json
Data written to ml.json!
```

Note: To reiterate, because we mounted our current location as a folder called `"/data"` (`-v $PWD:/data`), and we made sure to write the output file to that location in the container (`gen_ml_data.py /data/ml.json`), then we get to keep the file after the container exits, and it shows up in our current location (`$PWD`).

Alas, there is one more issue to address. The new file is owned by root, simply because it is root who created the file

inside the container. This is one minor Docker annoyance that we run in to from time to time. The simplest fix is to use one more `docker run` flag (`-id`) to specify the user and group ID namespace that should be used inside the container.

```
[isp02]$ ls -l
total 4
-rw-r--r--. 1 root root 2098 Feb 21 22:39 ml.json
[isp02]$ rm ml.json
rm: remove write-protected regular file ml.json'? y
[isp02]$ docker run --rm -v $PWD:/data -u $(id -u):$(id -g) username/gen_ml_data:1.0 gen_
↪ml_data.py /data/ml.json
Data written to /data/ml.json!
[isp02]$ ls -l
total 4
-rw-r--r--. 1 wallen G-815499 2098 Feb 21 22:41 ml.json
```

EXERCISE

Spend a few minutes testing both containers. Be sure you can generate data with one container, then read in and analyze the same data with the other. Data needs to persist outside the containers in order to do this.

4.4.2 Write a Compose File

Docker compose works by interpreting rules declared in a YAML file (typically called `docker-compose.yml`). The rules we will write will replace the `docker run` commands we have been using, and which have been growing quite complex. For example, the commands we used to run our JSON parsing scripts in a container looked like the following:

```
[isp02]$ docker run --rm -v $PWD:/data -u $(id -u):$(id -g) username/gen_ml_data:1.0 gen_
↪ml_data.py /data/ml.json
[isp02]$ docker run --rm -v $PWD:/data username/ml_data_analysis:1.0 ml_data_analysis.py ↪
↪/data/ml.json
```

The above `docker run` commands can be loosely translated into a YAML file. Navigate to the folder that contains your Python scripts and Dockerfiles, then create a new empty file called `docker-compose.yml`:

```
[isp02]$ pwd
/home/wallen/coe-332/docker-exercise
[isp02]$ touch docker-compose.yml
[isp02]$ ls
docker-compose.yml  Dockerfile-analysis  Dockerfile-gen  gen_ml_data.py  ml_data_
↪analysis.py  test/
```

Next, open up `docker-compose.yml` with your favorite text editor and type / paste in the following text:

```
1  ---
2  version: "3"
3
4  services:
5    gen-data:
6      build:
7        context: ./
8        dockerfile: ./Dockerfile-gen
9      image: username/gen_ml_data:1.0
```

(continues on next page)

(continued from previous page)

```
10     volumes:
11         - ./test:/data
12     user: "827385:815499"
13     command: gen_ml_data.py /data/ml.json
14 analyze-data:
15     build:
16         context: ./
17         dockerfile: ./Dockerfile-analysis
18     image: username/ml_data_analysis:1.0
19     volumes:
20         - ./test:/data
21     command: ml_data_analysis.py /data/ml.json
22     ...
```

Warning: The highlighted lines above need to be edited with your username / userid / groupid in order for this to work. See instructions below.

The `version` key must be included and simply denotes that we are using version 3 of Docker compose.

The `services` section defines the configuration of individual container instances that we want to orchestrate. In our case, we define two called `gen-data` for the `gen_ml_data` functionality, and `analyze-data` for the `ml_data_analysis` functionality.

Each of those services is configured with its own Docker image, a mounted volume (equivalent to the `-v` option for `docker run`), a user namespace (equivalent to the `-u` option for `docker run`), and a default command to run.

Please note that the image name above should be changed to use your image. Also, the user ID / group ID are specific to `wallen` - to find your user and group ID, execute the Linux commands `id -u` and `id -g`.

Note: The top-level `services` keyword shown above is just one important part of Docker compose. Later in this course we will look at named volumes and networks which can be configured and created with Docker compose.

4.4.3 Running Docker Compose

The Docker compose command line too follows the same syntax as other Docker commands:

```
docker-compose <verb> <parameters>
```

Just like Docker, you can pass the `--help` flag to `docker-compose` or to any of the verbs to get additional usage information. To get started on the command line tools, try issuing the following two commands:

```
[isp02]$ docker-compose version
[isp02]$ docker-compose config
```

The first command prints the version of Docker compose installed, and the second searches your current directory for `docker-compose.yml` and checks that it contains only valid syntax.

To run one of these services, use the `docker-compose run verb`, and pass the name of the service as defined in your YAML file:


```
[isp02]$ ls test/      # currently empty
[isp02]$ docker-compose run gen-data
Data written to /data/ml.json!
[isp02]$ ls test/
ml.json              # new file!
[isp02]$ docker-compose run analyze-data
6004.5
Southern & Eastern
... etc.
```

Now we have an easy way to run our *ad hoc* services consistently and reproducibly. Not only does `docker-compose`.
`yaml` make it easier to run our services, it also represents a record of how we intend to interact with this container.

4.4.4 Essential Docker Compose Command Summary

Command	Usage
<code>docker-compose version</code>	Print version information
<code>docker-compose config</code>	Validate <code>docker-compose.yml</code> syntax
<code>docker-compose up</code>	Spin up all services
<code>docker-compose down</code>	Tear down all services
<code>docker-compose build</code>	Build the images listed in the YAML file
<code>docker-compose run</code>	Run a container as defined in the YAML file

4.4.5 Additional Resources

- [Docker Compose Docs](#)

If you don't have one already, please create an account on [Docker Hub](#), which we will use extensively in this class. (The username does not have to be the same as your GitHub username, but it will make things less confusing later on if the usernames match).

UNIT 5: INTRODUCTION TO APIS AND FLASK

In Unit 5, we will be introduced to Application Programming Interfaces (APIs). This introduction will form the foundation of our ultimate goal to create large, complex, Python-based applications that are accessible through the web. The particular Python web framework we will be working with most is called Flask.

5.1 Introduction to APIs

In this section, we will discuss Application Programming Interfaces (APIs) focusing on Web APIs, and in particular REST APIs. We will learn how to interact with APIs using Python scripts. After going through this module, students should be able to:

- Identify and describe Web APIs (including REST APIs).
- List the four most important HTTP verbs and define how they are used in REST APIs.
- Describe how URLs are used to represent objects in a REST API.
- Explore API endpoints provided by various websites, e.g., GitHub.
- Install the Python `requests` library, and use it to interact with a web API in a Python script, including making requests and parsing responses.

An Application Programming Interface (API) establishes the protocols and methods for one piece of a program to communicate with another. APIs are useful for:

- (1) Allowing larger software systems to be built from smaller components,
- (2) Allowing the same component/code to be used by different systems, and
- (3) Insulating consumers from changes to the implementation.

Some examples of APIs:

- In OOP languages, abstract classes provide the interface for all concrete classes to implement.
- Software libraries provide an external interface for consuming programs.
- Web APIs (or “web services”) provide interfaces for computer programs to communicate over the internet.

While a User Interface connects humans to computer programs, an API is an interface that connects one piece of software to another.

APIs:

- (1) Provide functionality to external software in the form of a contract that specifies the inputs that the consuming software must provide and the outputs that the API will produce from the inputs.
- (2) Conceal the implementation of this functionality from the consuming software so that changes can be made to the implementation without impacting consumers.

- (3) Provide errors when the consuming software doesn't fulfill the contract of the API or when unexpected circumstances are encountered.

We have already been working with APIs. For example, the Python `json` library presents us with an API for working with JSON data.

```
>>> import json
>>> dir(json)
['JSONDecodeError',
'JSONDecoder',
'JSONEncoder',
. . .
'codecs',
'decoder',
'detect_encoding',
'dump',
'dumps',
'encoder',
'load',
'loads',
'scanner']
```

We use `json.dumps()` to convert Python objects to JSON (string) data and we use `json.loads()` to convert JSON strings to Python objects.

```
>>> json.dumps({'a': 1})
'{"a": 1}'
>>> type(_)
str
>>> json.loads('{"a": 1}')
{'a': 1}
>>> type(_)
dict
```

We say that `dumps` and `loads` are part of the Python `json` API. In terms of the contract, we might say something like:

- `json.loads()` – This function accepts a single input (string, bytes or bytes array) representing a valid JSON document and returns the equivalent Python object.
 - It will raise a `JSONDecodeError` if the input is not a valid JSON document.
- `json.dumps()` – This function accepts a single input (a Python object) and serializes it to a string. The Python object must be JSON serializable.
 - It will raise a `TypeError` if the input is not JSON serializable.

5.1.1 Web APIs

In this course, we will be building Web APIs or HTTP APIs. These are interfaces that are exposed over HTTP, allowing them to be consumed by software running on different machines.

There are a number of advantages to Web-based APIs that we will use in this class:

- A Web API can be made accessible to any computer or application that can access the public internet. Alternatively, a Web API can be restricted to a private network.
- No software installation is required on the client's side to consume a Web API.

- Web APIs can change their implementation without clients knowing (or caring).
- Virtually every modern programming language provides one or more libraries for interacting with a Web API; thus, Web APIs are “programming language agnostic”.

5.1.2 HTTP - the Protocol of the Internet

HTTP (Hyper Text Transfer Protocol) is one way for two computers on the internet to communicate with each other. It was designed to enable the exchange of data (specifically, “hypertext”). In particular, our web browsers use HTTP when communicating with web servers running web applications. HTTP uses a message-based, **client-server model**: clients make requests to servers by sending a message, and servers respond by sending a message back to the client.

HTTP is an “application layer” protocol in the language of the Internet Protocols; it assumes a lower level transport layer protocol. While this can be swapped, in practice it is almost always TCP. The basics of the protocol are:

- Web resources are identified with URLs (Uniform Resource Locators). Originally, **resources** were just files/directories on a server, but today resources refer to more general objects.
- HTTP “verbs” represent actions to take on the resource. The most common verbs are GET, POST, PUT, and DELETE.
- A **request** is made up of a URL, an HTTP verb, and a message
- A **response** consists of a status code (numerical between 100-599) and a message. The first digit of the status code specifies the kind of response:
 - 1xx - informational
 - 2xx - success
 - 3xx - redirection
 - 4xx - error in the request (client)
 - 5xx - error fulfilling a valid request (server)

5.1.3 Web Page Examples

Open a browser window, type `https://github.com` into the address bar and hit go. We see the GitHub home page which looks something like this:

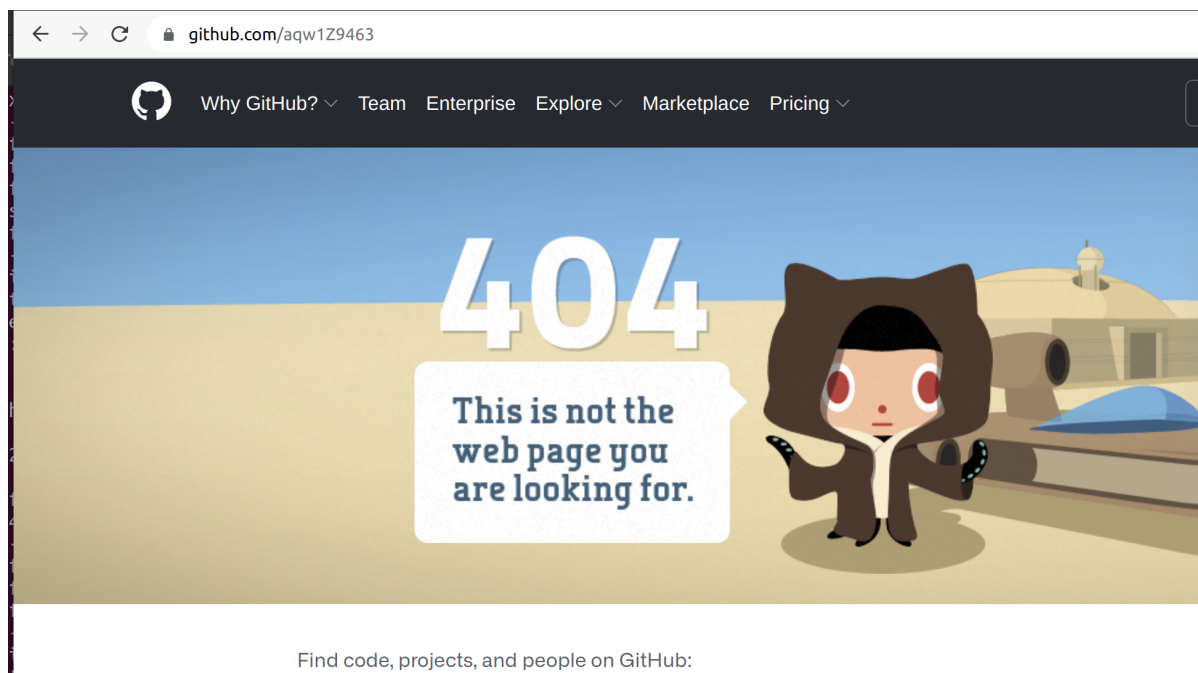
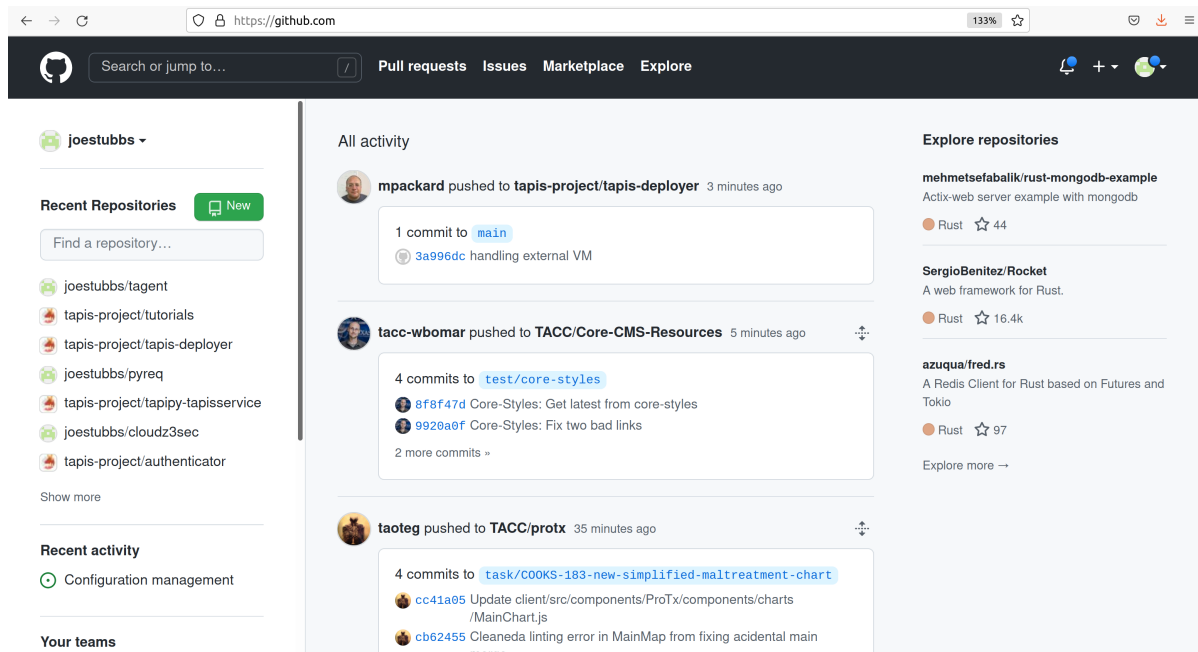
In fact, a multi-step process just occurred; here is a slightly simplified version of what happened:

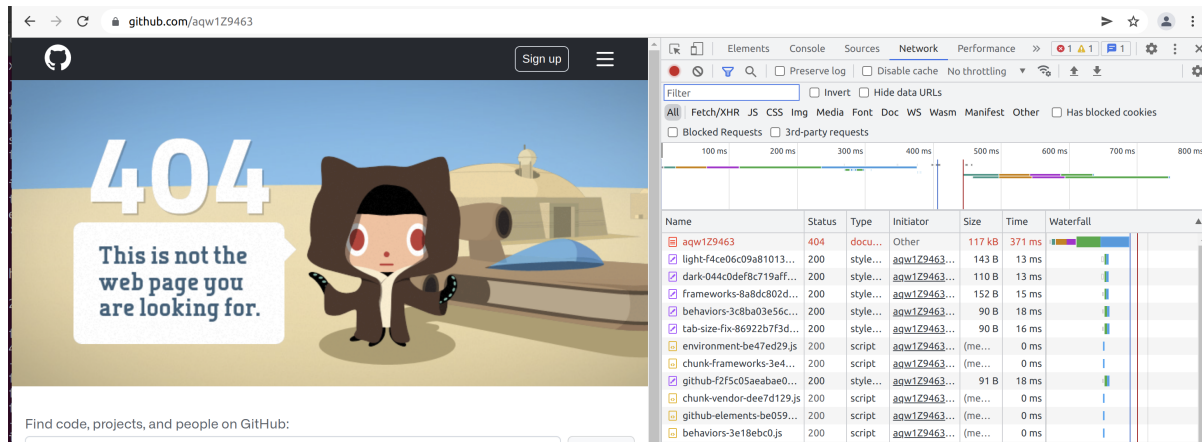
- (1) Your browser made an HTTP GET request to `https://github.com`.
- (2) A GitHub server received the request from your browser, formulated a response message containing the data (in HTML format) of your home page, with a 200 response code to indicate success.
- (3) Your browser received the response message from the GitHub server, and determined that the request was successful, due to the 200 response code.
- (4) It then drew the HTML message in the browser window.

If enter a URL that GitHub doesn’t recognize, we get a page that looks like this:

Most browsers have tools for determining what requests and responses were made. For example, in Chrome, we can use “More Tools -> Developer Tools” from the Customize and Control menu (the three dots in the top-right corner), to open up a panel for introspecting the requests being made.

If we click the “Network” tab and try our request again, we will see something like this:





The top row in red represents the request to <https://github.com/aqw1Z9463> and it shows the status code of the response was 404.

5.1.4 REST APIs - Overview

REST (Representational State Transfer) is a way of building APIs for computer programs on the internet leveraging HTTP. In other words, a program on computer 1 interacts with a program on computer 2 by making an HTTP request to it and receiving HTTP responses.

The basic idea with REST is to associate objects in the application domain with URLs, and to use HTTP verbs to represent the actions we want to take on the objects. A REST API has a **base URL** from which all other URLs in that API are formed. For example, the base URL for the GitHub REST API which will look at in more detail momentarily is <https://api.github.com/>.

The other URLs in the API are then “collections”, typically represented by a plural noun, following the base URL; e.g.:

```
<base_url>/users
<base_url>/files
<base_url>/programs
```

Or they are specific items in a collection, represented by an identifier following the collection name, e.g.:

```
<base_url>/users/12345
<base_url>/files/test.txt
<base_url>/programs/myapplication
```

Or subcollections or items in subcollections, e.g.:

```
<base_url>/companies/<company_id>/employees
<base_url>/companies/<company_id>/employees/<employee_id>
```

As mentioned, the HTTP verbs represent “operations” or actions that can be taken on the resources:

- GET - list items in a collection or retrieve a specific item in the collection
- POST - create a new item in the collection based on the description in the message
- PUT - replace an item in a collection with the description in the message
- DELETE - delete an item in a collection

Thus,

- GET <base_url>/users would list all users.
- POST <base_url>/users would create a new user.
- PUT <base_url>/users/12345 would update user 12345.

The combination of an HTTP verb and URL (path) is called an **endpoint** in an API. A REST API is typically comprised of many endpoints.

Note that not all HTTP verbs make sense for all URLs. For example, an API would probably not include a PUT <base_url>/users endpoint, because semantically, that would mean updating the entire list of users.

Response messages often make use of some data serialization format standard such as JSON, CSV or XML.

5.1.5 REST APIs - Additional Simple Examples

Virtually every application domain can be mapped into a REST API architecture. Some examples may include:

Articles in a collection (e.g., on a blog or wiki) with author attributes:

```
<base_url>/articles
<base_url>/articles/<id>
<base_url>/articles/<id>/authors
```

Properties in a real estate database with associated purchase history:

```
<base_url>/properties
<base_url>/properties/<id>
<base_url>/properties/<id>/purchases
```

A catalog of countries, cities and neighborhoods:

```
<base_url>/countries
<base_url>/countries/<country_id>/cities
<base_url>/countries/<country_id>/cities/<city_id>/neighborhoods
```

5.1.6 REST APIs - A Real Example

We have been using GitHub to host our class code repositories. It turns out GitHub provides an HTTP API that is architected using REST (for the most part). We're going to explore the GitHub API.

To begin, open a web browser and navigate to <https://api.github.com>

You will see something like this:

```
{
  "current_user_url": "https://api.github.com/user",
  "current_user_authorizations_html_url": "https://github.com/settings/connections/
↪applications{/client_id}",
  "authorizations_url": "https://api.github.com/authorizations",
  "code_search_url": "https://api.github.com/search/code?q={query}{&page,per_page,sort,
↪order}",
  "commit_search_url": "https://api.github.com/search/commits?q={query}{&page,per_page,
↪sort,order}",
```

(continues on next page)

(continued from previous page)

```

"emails_url": "https://api.github.com/user/emails",
"emojis_url": "https://api.github.com/emojis",
"events_url": "https://api.github.com/events",
"feeds_url": "https://api.github.com/feeds",
"followers_url": "https://api.github.com/user/followers",
"following_url": "https://api.github.com/user/following{/target}",
"gists_url": "https://api.github.com/gists{/gist_id}",
"hub_url": "https://api.github.com/hub",
"issue_search_url": "https://api.github.com/search/issues?q={query}{&page,per_page,
↪sort,order}",
"issues_url": "https://api.github.com/issues",
"keys_url": "https://api.github.com/user/keys",
"label_search_url": "https://api.github.com/search/labels?q={query}&repository_id=
↪{repository_id}{&page,per_page}",
"notifications_url": "https://api.github.com/notifications",
"organization_url": "https://api.github.com/orgs/{org}",
"organization_repositories_url": "https://api.github.com/orgs/{org}/repos{?type,page,
↪per_page,sort}",
"organization_teams_url": "https://api.github.com/orgs/{org}/teams",
"public_gists_url": "https://api.github.com/gists/public",
"rate_limit_url": "https://api.github.com/rate_limit",
"repository_url": "https://api.github.com/repos/{owner}/{repo}",
"repository_search_url": "https://api.github.com/search/repositories?q={query}{&page,
↪per_page,sort,order}",
"current_user_repositories_url": "https://api.github.com/user/repos{?type,page,per_
↪page,sort}",
"starred_url": "https://api.github.com/user/starred{/owner}/{repo}",
"starred_gists_url": "https://api.github.com/gists/starred",
"topic_search_url": "https://api.github.com/search/topics?q={query}{&page,per_page}",
"user_url": "https://api.github.com/users/{user}",
"user_organizations_url": "https://api.github.com/user/orgs",
"user_repositories_url": "https://api.github.com/users/{user}/repos{?type,page,per_
↪page,sort}",
"user_search_url": "https://api.github.com/search/users?q={query}{&page,per_page,sort,
↪order}"
}

```

This should look familiar – it’s a JSON document, and it describes various collections of endpoints in the GitHub API. For example, we see:

- "events_url": "https://api.github.com/events", – Work with GitHub events
- "organization_url": "https://api.github.com/orgs/{org}", – Work with GitHub orgs
- "repository_url": "https://api.github.com/repos/{owner}/{repo}", – Work with GitHub repos

Many of the endpoints within the GitHub API require *authentication*, i.e., that the requesting application prove its identity – we’ll ignore this topic for now and just work with the endpoints that do not require authentication.

Let’s discover what the GitHub API can tell us about TACC’s GitHub organization, which is just called tacc.

EXERCISE

Based on the information above, how would we retrieve information about the TACC GitHub organization from the API? What HTTP verb and URL would we use?

SOLUTION

We see that the “organization_url” is defined to be “<https://api.github.com/orgs/{org}>”. The use of the {org} notation is common in API documentation – it indicates a variable to be substituted with a value. In this case, we should substitute tacc for {org}, as that is the organization we are interested in.

Since we want to retrieve (or list) information about the TACC organization, the HTTP verb we want to use is GET.

We can use the browser to make this request, as before. If we enter <https://api.github.com/orgs/tacc> into the URL bar, we should see:

```
{
  "login": "TACC",
  "id": 840408,
  "node_id": "MDEyOk9yZ2FuaXphdGlvbjg0MDQwOA==",
  "url": "https://api.github.com/orgs/TACC",
  "repos_url": "https://api.github.com/orgs/TACC/repos",
  "events_url": "https://api.github.com/orgs/TACC/events",
  "hooks_url": "https://api.github.com/orgs/TACC/hooks",
  "issues_url": "https://api.github.com/orgs/TACC/issues",
  "members_url": "https://api.github.com/orgs/TACC/members{/member}",
  "public_members_url": "https://api.github.com/orgs/TACC/public_members{/member}",
  "avatar_url": "https://avatars.githubusercontent.com/u/840408?v=4",
  "description": "",
  "name": "Texas Advanced Computing Center",
  "company": null,
  "blog": "http://www.tacc.utexas.edu",
  "location": "Austin, TX",
  "email": null,
  "twitter_username": null,
  "is_verified": false,
  "has_organization_projects": true,
  "has_repository_projects": true,
  "public_repos": 152,
  "public_gists": 0,
  "followers": 0,
  "following": 0,
  "html_url": "https://github.com/TACC",
  "created_at": "2011-06-09T16:47:08Z",
  "updated_at": "2021-04-07T17:34:55Z",
  "type": "Organization"
}
```

5.1.7 Using Python to Interact with Web APIs

Viewing API response messages in a web browser provides limited utility. We can interact with Web APIs in a much more powerful and programmatic way using the Python `requests` library.

First install the `requests` library in your local site-packages on the ISP server using `pip3`:

```
[isp02]$ pip3 install --user requests
...
Successfully installed requests-2.25.1
```

You might test that the install was successful by trying to import the library in the interactive Python interpreter:

```
[isp02]$ python3
Python 3.6.8 (default, Aug  7 2019, 17:28:10)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-39)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import requests
>>>
```

The basic usage of the `requests` library is as follows:

```
>>> # make a request: typical format
>>> # response = requests.<method>(url=some_url, data=some_message, <other options>)
>>>
>>> # e.g. try:
>>> response = requests.get(url='https://api.github.com/orgs/tacc')
>>>
>>> # return the status code:
>>> response.status_code
>>>
>>> # return the raw content
>>> response.content
>>>
>>> # return a Python list or dictionary from the response message
>>> response.json()
```

EXERCISE

Let's use `requests` to explore the GitHub API. Write functions to return the following:

- Given a GitHub organization id, retrieve all information about the organization. Return the information as a Python dictionary.
- Given a GitHub organization id, retrieve a list of all of the members of the organization. Return the list of members as a Python list of strings, where each string contains the member's `login` (i.e., GitHub username) attribute.
- Given a GitHub organization id, return a list of repositories controlled by the organization. Return the list of repositories as a Python list of strings, where each string contains the repository `full_name` attribute.

5.2 Introduction to Flask

In this section, we will get a brief introduction into Flask, the Python web framework, including how to set up a REST API with multiple routes (URLs). After going through this module, students should be able to:

- Install the Python Flask library and import it into a Python program.
- Define and implement various “routes” or API endpoints in a Flask Python program.
- Run a local Flask development server.
- Use curl to test routes defined in their Flask program when the local Flask development server is running.

Flask is a Python library and framework for building web servers. Some of the defining characteristics of Flask make it a good fit for this course:

- Flask is small and lightweight - relatively easy to use and get setup initially
- Flask is robust - a great fit for REST APIs and **microservices**
- Flask is performant - when used correctly, it can handle the traffic of sites with 100Ks of users

5.2.1 What is a Microservice?

Microservices - also known as the microservice architecture - is an architectural style that structures an application as a collection of services that are:

- Highly maintainable and testable
- Loosely coupled
- Independently deployable
- Organized around business capabilities

The microservice architecture enables the continuous delivery/deployment of large, complex applications. It also enables an organization to evolve its technology stack. Many heavily-used, well-known sites use microservices including Netflix, Amazon, and eBay.

There is a great article on DevTeam.Space [about microservices](#).

5.2.2 Setup and Installation

The Flask library is not part of the Python standard library but can be installed with standard tools like pip3. In addition to making Flask available to import into a Python program, it will also expose some new command line tools. On the class server, perform the following:

```
[isp02]$ pip3 install --user flask
...
Successfully installed flask-1.1.2

[isp02]$ flask --help
Usage: flask [OPTIONS] COMMAND [ARGS]...
```

A general utility script for Flask applications.

Provides commands from Flask, extensions, and the application. Loads the application defined in the FLASK_APP environment variable, or from a

(continues on next page)

(continued from previous page)

wsgi.py file. Setting the FLASK_ENV environment variable to 'development' will enable debug mode.

```
> export FLASK_APP=hello.py
> export FLASK_ENV=development
> flask run
```

Options:

```
--version  Show the flask version
--help     Show this message and exit.
```

Commands:

```
routes  Show the routes for the app.
run      Run a development server.
shell    Run a shell in the app context.
```

Tip: If you aren't already using a virtual environment to help manage your Python libraries, now is a [good time to start!](#)

5.2.3 A Hello World Flask App

In a new directory on the class server, create a file called `app.py` and open it for editing. Enter the following lines of code:

```
1 from flask import Flask
2
3 app = Flask(__name__)
4
5 # the next statement should usually appear at the bottom of a flask app
6 if __name__ == '__main__':
7     app.run(debug=True, host='0.0.0.0')
```

On the first line, we are importing the Flask class.

On the third line, we create an instance of the Flask class (called `app`). This so-called “Flask application” object holds the primary configuration and behaviors of the web server.

Finally, the `app.run()` method launches the development server. The `debug=True` option tells Flask to print verbose debug statements while the server is running. The `host=0.0.0.0` option instructs the server to listen on all network interfaces; basically this means you can reach the server from inside and outside the host VM.

5.2.4 Run the Flask App

There are two main ways of starting the Flask service. For now, we recommend you start the service using a unique port number. The `-p 5000` indicates that Flask is running on port 5000. You will need to use your own assigned port.

Warning: Check Slack or ask the instructors which port you should use. Trying to run two Flask apps on the same port will not work.

```
[isp02]$ export FLASK_APP=app.py
[isp02]$ export FLASK_ENV=development
[isp02]$ flask run -p 5000
* Serving Flask app "app.py" (lazy loading)
* Environment: development
* Debug mode: on
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 182-299-771
```

That's it! We now have a server up and running. Some notes on what is happening:

- Note that the program took over our shell; we could put it in the background, but for now we want to leave it in the foreground. (Multiple PIDs are started for the Flask app when started in daemon mode; to get them, find all processes listening on the port 5000 socket with `lsof -i:5000`).
- If we make changes to our Flask app while the server is running in development mode, the server will detect those changes automatically and “reload”; you will see a log to the effect of `Detected change in <file>`.
- We can stop the program with `Ctrl+C` just like any other (Python) program.
- If we stop our Flask programs, the server will no longer be listening and our requests will fail.

Next we can try to talk to the server using `curl`. Note this line:

```
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

That tells us our server is listening on the `localhost` - `127.0.0.1`, and on the default Flask port, port `5000`.

Ports Basics

Ports are a concept from networking that allow multiple services or programs to be running at the same time, listening for messages over the internet, on the same computer.

- For us, ports will always be associated with a specific IP address. In general, we specify a port by combining it with an IP separated by a colon (`:`) character. For example, `129.114.97.16:5000`.
- One and only one program can be listening on a given port at a time.
- Some ports are designated for specific activities; Port 80 is reserved for HTTP, port 443 for HTTPS (encrypted HTTP), but other ports can be used for HTTP/HTTPS traffic.

curl Basics

You can think of `curl` as a command-line version of a web browser: it is just an HTTP client.

- The basic syntax is `curl <some_base_url>:<some_port>/<some_url_path>`. This will make a GET request to the URL and port print the message response.
- Curl will default to using port 80 for HTTP and port 443 for HTTPS.
- You can specify the HTTP verb to use with the `-X` flag; e.g., `curl -X GET <some_url>` (though `-X GET` is redundant because that is the default verb).
- You can set “verbose mode” with the `-v` flag, which will then show additional information such as the headers passed back and forth (more on this later).

5.2.5 Make a Request

Because the terminal window running your Flask app is currently locked to that process, the simplest thing to do is open up a new terminal and SSH into the class server again.

To make a request to your Flask app, type the following in the new terminal:

```
[isp02]$ curl 127.0.0.1:5000
- or -
[isp02]$ curl localhost:5000
```

You should see the following response:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>404 Not Found</title>
<h1>Not Found</h1>
<p>The requested URL was not found on the server. If you entered the URL manually,
↪ please check your spelling and try again.</p>
```

Our server is sending us HTML! It's sending a 404 that it could not find the resource we requested. Although it appears to be an error (and technically it is), this is evidence that the Flask server is running successfully. It's time to add some routes.

Note: Only one Flask app can be associated with each port. The default port above (5000) is an example. Please make sure to run your Flask server on the port assigned to you (`flask run -p 50xx`). You can curl your own port number, or you can curl other people's Flask servers by subbing in their port number.

5.2.6 Routes in Flask

In a Flask app, you define the URLs in your application using the `@app.route` decorator. Specifications of the `@app.route` decorator include:

- Must be placed on the line before the declaration of a Python function.
- Requires a string argument which is the path of the URL (not including the base URL)
- Takes an argument `methods` which should be a list of strings containing the names of valid HTTP methods (e.g. GET, POST, PUT, DELETE)

When the URL + HTTP method combination is requested, Flask will call the decorated function.

Tangent: What is a Python Decorator?

A decorator is a function that takes another function as an input and extends its behavior in some way. The decorator function itself must return a function which includes a call to the original function plus the extended behavior. The typical structure of a decorator is as follows:

```
1 def my_decorator(some_func):
2
3     def func_to_return():
4
5         # extend the behavior of some_func by doing some processing
6         # before it is called (optional)
7         do_something_before()
8
9         # call the original function
10        some_func(*args, **kwargs)
11
12        # extend the behavior of some_func by doing some processing
13        # after it is called (optional)
14        do_something_after()
15
16    return func_to_return
```

As an example, consider this test program:

```
1 def print_dec(f):
2     def func_to_return(*args, **kwargs):
3         print("args: {}; kwargs: {}".format(args, kwargs))
4         val = f(*args, **kwargs)
5         print("return: {}".format(val))
6         return val
7     return func_to_return
8
9 @print_dec
10 def foo(a):
11     return a+1
12
13 result = foo(2)
14 print("Got the result: {}".format(result))
```

Our @print_dec decorator gets executed automatically when we call foo(2). Without the decorator, the final output would be:

```
Got the result: 3
```

By using the decorator, however, the final output is instead:

```
args: (2,); kwargs: {}
return: 3
Got the result: 3
```


5.2.7 Define the Hello World Route

The original Flask app we wrote above (in `app.py`) did not define any routes. Let's define a "hello world" route for the base URL. Meaning if someone were to curl against the base URL (/) of our server, we would want to return the message "Hello, world!". To do so, add the following lines to your `app.py` script:

```

1  from flask import Flask
2
3  app = Flask(__name__)
4
5  @app.route('/', methods=['GET'])
6  def hello_world():
7      return 'Hello, world!\n'
8
9  # the next statement should usually appear at the bottom of a flask app
10 if __name__ == '__main__':
11     app.run(debug=True, host='0.0.0.0')
```

The `@app.route` decorator on line 5 is expecting GET requests at the base URL /. When it receives such a request, it will execute the `hello_world()` function below it.

In your active SSH terminal, execute the curl command again (you may need to restart the Flask app); you should see:

```
[isp02]$ curl localhost:5000/
Hello, world!
```

5.2.8 Routes with URL Parameters

Flask makes it easy to create routes (or URLs) with variables in the URL. The variable name simply must appear in angled brackets (<>) within the `@app.route()` decorator statement; for example the following would grant the function below it access to a variable called `year`:

```
@app.route('/<year>')
```

In the next example, we extend our `app.py` Flask app by adding a route with a variable (<name>):

```

1  from flask import Flask
2
3  app = Flask(__name__)
4
5  @app.route('/', methods=['GET'])
6  def hello_world():
7      return 'Hello, world!\n'
8
9  @app.route('/<name>', methods=['GET'])
10 def hello_name(name):
11     return f'Hello, {name}!\n'
12
13 # the next statement should usually appear at the bottom of a flask app
14 if __name__ == '__main__':
15     app.run(debug=True, host='0.0.0.0')
```

Now, the Flask app supports multiple routes with different functionalities:

```
[isp02]$ curl localhost:5000/  
Hello, world!  
[isp02]$ curl localhost:5000/joe  
Hello, joe!  
[isp02]$ curl localhost:5000/jane  
Hello, jane!
```

EXERCISE

Let's utilize the meteorite landing data from the Advances Containers section of Unit 4 to define a somewhat more interesting route. We'll create a route that allows a user to download the entire dataset over HTTP.

We'll add one new route function. Consider the following?

- What should the name of our function be?
- What URL path should it respond to?
- What HTTP verb(s) should it handle?

Once those questions are answered, we'll need to actually implement the new route function. What will we need to do to implement the function? The implementation will require two steps:

- 1) Read the data into Python from the JSON file.
- 2) Return the result of step 1).

Finally, once we have implemented the function, let's test it using `curl`.

5.3 Advanced Flask

We continue using Flask in this module with a look at more complex endpoints and data retrieval functions for our REST API. After going through this module, students should be able to:

- Identify valid and invalid Flask route return types
- Convert unsupported types (e.g. `list`) to valid Flask route return types
- Extract Content-Type and other headers from Flask route responses
- Add query parameters to GET requests, and extract their values inside Flask routes
- Deal with errors from user-supplied input to an API and handle Python exceptions

Note: We will continue to do our work on the isp02 VM. Like last time, it will be helpful for you to have two SSH terminals open to isp02 at the same time so you can run your Flask application in one terminal and test it in the other.

5.3.1 Defining the URLs of Our API

One of our first goals for our API will be to provide an interface to a dataset. Since the URLs in a REST API are defined by the “nouns” or collections of the application domain, we can use a noun that represents our data.

For example, suppose we have the following dataset that represents the number of students earning an undergraduate degree for a given year:

```
def get_data():
    return [ {'id': 0, 'year': 1990, 'degrees': 5818},
             {'id': 1, 'year': 1991, 'degrees': 5725},
             {'id': 2, 'year': 1992, 'degrees': 6005},
             {'id': 3, 'year': 1993, 'degrees': 6123},
             {'id': 4, 'year': 1994, 'degrees': 6096} ]
```

In this case, one collection described by the data is “degrees”. So, let’s define a route, `/degrees`, that by default returns all of the data points.

EXERCISE 1

Create a new file, `degrees_api.py` to hold a Flask application then do the following:

- 1) Import the Flask class and instantiate a Flask application object.
- 2) Add code so that the Flask server is started when this file is executed directly by the Python interpreter.
- 3) Copy the `get_data()` method above into the application script.
- 4) Add a route (`/degrees`) which responds to the HTTP GET request and returns the complete list of data returned by `get_data()` as a Python string. **Hint:** You won’t be able to return a Python list directly from your route function – consider casting the list to a string using the `str()` function.

In a separate Terminal use `curl` to test out your new route. Does it work as expected?

Tip: Refer back to the [Intro to Flask material](#) if you need help remembering the boiler-plate code.

5.3.2 Responses in Flask

If you tried to return the list object directly in your route function definition, you got an error when you tried to request it with `curl`. Something like:

```
TypeError: The function did not return a valid response
```

Flask allows you three options for creating responses:

- 1) Return a string (`str`) object
- 2) Return a dictionary (`dict`) object
- 3) Return a tuple (`tuple`) object
- 4) Return a `flask.Response` object

Some notes:

- Option 1 is good for text or html such as when returning a web page or text file.
- Option 2 is good for returning rich information in JSON-esque format.

- Option 3 is good for returning a list of data using a special type of Python list - a `tuple` - which is ordered and unchangeable.
- Option 4 gives you the most flexibility, as it allows you to customize the headers and other aspects of the response.

For our REST API, we will want to return JSON-formatted data. We will use a special Flask method to convert our list to JSON - `flask.jsonify`. (More on this later.)

Tip: Refer back to the [Working with JSON material](#) for a primer on the JSON format and relevant JSON-handling methods.

EXERCISE 2

Try doing this exercise in a Python (or `ipython`) shell.

- 1) Serialize the list returned by the `get_data()` method above into a JSON-formatted string using the Python `json` library. Verify that the type returned is a string.
- 2) Next, deserialize the string returned in part 1 by using the `json` library to decode it. Verify that the result equals the original list.

5.3.3 Returning JSON (and Other Kinds of Data)

You probably are thinking at this point we can fix our solution to the first **Exercise** by using the `json` library (which function?). Let's try that and see what happens:

EXERCISE 3

Update your code from the first Exercise to use the `json` library to return a properly formatted JSON string.

Then, with your API server running in one window, open a Python3 interactive session in another window and:

- Make a GET request to your `/degrees` URL and capture the response in a variable, say `r`
- Verify that `r.status_code` is what you expect (what do you expect it to be?)
- Verify that `r.content` is what you expect.
- Use `r.json()` to decode the response and compare the type to that of `r.content`.

5.3.4 HTTP Content Type Headers

Requests and responses have **headers** which describe additional metadata about them. Headers are **key: value** pairs (much like dictionary entries). The **key** is called the header name and the **value** is the header value.

There are many pre-defined headers for common metadata such as specifying the size of the message (**Content-Length**), the domain the server is listening on (**Host**), and the type of content included in the message (**Content-Type**).

We can use `curl` or the python `requests` library to see all of the headers returned on a response from our flask server. Let's try it.

EXERCISE 4

- 1) Use `curl` to make a GET request to your `/degrees` endpoint and pass the `-v` (for “verbose”) option. This will show you additional information, including the headers. Note that with `-v`, `curl` shows headers on both the request and the response. Request headers are lines that start with a `>` while response headers are lines that start with a `<`.
- 2) Use `curl` again to make the same request, but this time pass the `--head` option instead of the `-v`; this will show you **only** the headers being returned in the response.
- 3) Inside a Python shell, use `requests` to make the same GET request to your `/degrees` endpoint, and capture the result in a variable, `r`. Inspect the `r.header` attribute. What is the type of `r.headers`?

```
curl localhost:5000/degrees -v

* About to connect() to localhost port 5000 (#0)
* Trying ::1...
* Connection refused
* Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 5000 (#0)
> GET /degrees HTTP/1.1
> User-Agent: curl/7.29.0
> Host: localhost:5000
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Content-Type: text/html; charset=utf-8
< Content-Length: 210
< Server: Werkzeug/2.0.3 Python/3.6.8
< Date: Fri, 04 Mar 2022 01:12:34 GMT
```

```
In [1]: import requests

In [2]: r = requests.get('http://127.0.0.1:5000/degrees')

In [3]: r.headers
Out[3]: {'Content-Type': 'text/html; charset=utf-8', 'Content-Length': '210', 'Server':
↪ 'Werkzeug/2.0.3 Python/3.6.8', 'Date': 'Fri, 04 Mar 2022 01:21:41 GMT'}
```

We see that we are sending a `Content-type` of `'text/html'`. In some ways, that is true, but we can do better; we can tell the client we are sending `json` data.

Media Type (or Mime Type)

The allowed values for the `Content-Type` header are the defined **media types** (formerly, **mime types**). The main thing you want to know about media types are that they:

- Consist of a type and subtype
- The most common types are application, text, audio, image, and multipart
- The most common values (type and subtype) are application/json, application/xml, text/html, audio/mpeg, image/png, and multipart/form-data

Content Types in Flask

The Flask library has the following built-in conventions you want to keep in mind:

- When returning a string as part of a route function in Flask, a **Content-Type** of `text/html` is returned.
- To convert a Python object to a JSON-formatted string **and** set the content type properly, use the `flask.jsonify()` function.

For example, the following code will convert the list to a JSON string and return a content type of `application/json`:

```
return flask.jsonify(['a', 'b', 'c'])
```

EXERCISE 5

Use the `flask.jsonify()` method to update your code from Exercise 1. Then:

- Validate that your `/degrees` endpoint works as expected by using the `requests` library to make an API request and check that the `.json()` method works as expected on the response.
- Use the `.headers()` method on the response to verify the **Content-Type** is what you expect.

5.3.5 Query Parameters

The HTTP specification allows for parameters to be added to the URL in form of `key=value` pairs. Query parameters come after a `?` character and are separated by `&` characters; for example, the following request to a hypothetical API:

```
GET https://api.example.com/degrees?limit=3&offset=2
```

passes two query parameters: `limit=3` and `offset=2`. Note that the URL path in the example above is still `/degrees`; that is, the `?` character terminates the URL path, and any characters that follow create the query parameter set for the request.

In REST architectures, query parameters are often used to allow clients to provide additional, optional arguments to the request.

Common uses of query parameters in RESTful APIs include:

- **Pagination:** specifying a specific page of results from a collection
- **Search terms:** filtering the objects within a collection by additional search attributes
- **Other parameters** that might apply to most if not all collections such as an ordering attribute (`ascending` vs `descending`)

Extracting Query Parameters in Flask

Flask makes the query parameters available on the `request.args` object, which is a “dictionary-like” object. To work with the query parameters supplied on a request, you must import the Flask `request` object, and use the `args.get` method to extract the passed query parameter into a variable.

Note: The `flask.request` object is different from the Python3 `requests` library we used to make http requests. the `flask.request` object represents the incoming request that our flask application server has received from the client.

```

1 from flask import Flask, request
2
3 @app.route('/degrees', methods=['GET'])
4 def degrees():
5     start = request.args.get('start')
6     # additional code...

```

The start variable will be the value of the start parameter, if one is passed, or it will be None otherwise:

```
GET https://api.example.com/degrees?start=2
```

Note: request.args.get() will always return a string, regardless of the type of data being passed in.

Let's use this idea to update our degrees_api to only return the years starting from the start query parameter year, if that parameter is provided.

```

@app.route('/degrees', methods=['GET'])
def degrees():
    d = get_data()
    start = int(request.args.get('start', 0))
    return flask.jsonify(d)

```

5.3.6 Error Handling

This brings up the topic of error handling. What happens if the user sends a value for the start query parameter that isn't an integer? We can test it ourselves.

```
[isp02]$ curl 127.0.0.1:5000/degrees?start=abc
```

If we try this we get some nasty stuff that ends with a traceback, like this:

```

Traceback (most recent call last):
File "/home/jstubbbs/.local/lib/python3.6/site-packages/flask/app.py", line 2091, in __
↳ call__
    return self.wsgi_app(environ, start_response)
File "/home/jstubbbs/.local/lib/python3.6/site-packages/flask/app.py", line 2076, in wsgi_
↳ app
    response = self.handle_exception(e)
File "/home/jstubbbs/.local/lib/python3.6/site-packages/flask/app.py", line 2073, in wsgi_
↳ app
    response = self.full_dispatch_request()
File "/home/jstubbbs/.local/lib/python3.6/site-packages/flask/app.py", line 1518, in full_
↳ dispatch_request
    rv = self.handle_user_exception(e)
File "/home/jstubbbs/.local/lib/python3.6/site-packages/flask/app.py", line 1516, in full_
↳ dispatch_request
    rv = self.dispatch_request()
File "/home/jstubbbs/.local/lib/python3.6/site-packages/flask/app.py", line 1502, in _
↳ dispatch_request
    return self.ensure_sync(self.view_functions[rule.endpoint])(**req.view_args)
File "/home/jstubbbs/coe332-sp22/flask_2/degrees_api_2.py", line 31, in degrees3

```

(continues on next page)

(continued from previous page)

```
start = int(request.args.get('start', 0))
ValueError: invalid literal for int() with base 10: 'abc'
```

Checking User Input

If we get a request like this, in the best case, the user didn't understand what kind of data to put in the `start` query parameter; in the worst case, they were intentionally trying to send our program bad data to break it. We should always be very careful with user-supplied data and make sure it contains the kind of data we expect.

So, what is it we expect from the `start` query parameter? At a minimum, it needs to be some kind of integer data, because we are casting it to the `int` type. Therefore, at a minimum, we should check if it is an integer.

We can use the Python `isnumeric()` method on a Python string to test whether a string contains non-negative integer data.

Let's try some examples in the Python shell:

```
>>> '123'.isnumeric()
True
>>> 'abc'.isnumeric()
False
>>> '1.23'.isnumeric()
False
>>> '-1'.isnumeric()
False
```

Now, let's fix our route function; we can check if it is numeric before casting to an `int`. If it is not numeric, we can return an error message to the user.

```
1 @app.route('/degrees', methods=['GET'])
2 def degrees():
3     d = get_data()
4     start = request.args.get('start', 0)
5     if not start.isnumeric():
6         return 'Invalid start value; start must be numeric.'
7     start = int(start)
8     return flask.jsonify(d[start:])
```

Exceptions

Using the `isnumeric()` function allowed us to check for invalid user input in the specific case above, but Python provides a far more general and powerful error handling capability, called Exceptions, that we will discuss next.

In Python, exceptions are the mechanism one typically uses to communicate and deal with run-time errors. Exceptions are different from syntax errors where, in general, there is no hope of the code working. Exceptions occur with statements that are syntactically correct but nonetheless generate some kind of error at runtime. Typically, the program can recover from these types of errors.

In Python, exceptions are instances of the class `Exception` or a child class. We say that a statement *generates* or *raises* an exception.

Some common situations that generate exceptions are:

- Trying to open a file that does not exist raises a `FileNotFoundError`.

- Trying to divide by zero raises a `ZeroDivisionError`.
- Trying to access a list at an index beyond its length raises an `IndexError`.
- Trying to use an object of the wrong type in a function raises a `TypeError` (for example, trying to call `json.dumps()` with an object that is not of type `str`.)
- Trying to use an object with the wrong kind of value in a function raises a `ValueError` (for example, calling `int('abc')`.)
- Trying to access a non-existent attribute on an object raises an `AttributeError` (a special case is accessing a null/uninitialized object, resulting in the dreaded `AttributeError: 'NoneType' object has no attribute 'foo'` error.)

Handling Exceptions

If a statement we execute in our code, such as a call to the `int()` function to cast an object to an integer, could raise an exception, we can handle the exception by using the `try...except...` statement. It works like this:

```
try:
    # execute some statements that could raise an exception...
    f(x, y, z)
except ExceptionType1 as e:
    # do something if the exception was of type ExceptionType1...
except ExceptionType2 as e:
    # do something if the exception was of type ExceptionType2...

# . . . additional except blocks . . .

finally:
    # do something regardless of whether an exception was raised or not.
```

A few notes:

- If a statement(s) within the `try` block does not raise an exception, the `except` blocks are skipped.
- If a statement within the `try` block does raise an exception, Python looks at the `except` blocks for the first one matching the type of the exception raised and executes that block of code.
- The `finally` block is optional but it executes regardless of whether an exception was raised by a statement or not.
- The `as e` clause puts the exception object into a variable (`e`) that we can use.
- The use of `e` was arbitrary; we could choose to use any other valid variable identifier.
- We can also leave off the `as e` part altogether if we don't need to reference the exception object in our code.

Here's how we could deal with an invalid `start` parameter provided by the user using exceptions:

```
try:
    start = int(start)
except ValueError:
    # return some kind of error message...

# at this point in the code, we know the int(start) "worked" and so we are safe
# to use it as an integer..
```

Exception Hierarchy

Exceptions form a class hierarchy with the base `Exception` class being at the root. So, for example:

- `FileNotFoundError` is a type of `OSError` as is `PermissionError`, which is raised in case the attempted file access is not permitted by the OS due to lack of permissions.
- `ZeroDivisionError` and `OverflowError` are instances of `ArithmeticError`, the latter being raised whenever the result of a calculation exceeds the limits of what can be represented (try running `2.**5000` in a Python shell).
- Every built-in Python exception is of type `Exception`.

Therefore, we could use any of the following to deal with a `FileNotFoundError`:

- `except FileNotFoundError`
- `except OSError`
- `except Exception`

Here are some best practices to keep in mind for handling exceptions:

- Put a minimum number of statements within a `try` block so that you can detect which statement caused the error.
- Similarly, put the most specific exception type in the `except` block that is appropriate so that you can detect exactly what went wrong. Using `except Exception...` should be seen as a last resort because an `Exception` could be any kind of error.

Here is the full code for our route function with exception handling.

```
1 @app.route('/degrees', methods=['GET'])
2 def degrees():
3     d = get_data()
4     start = request.args.get('start')
5     if start:
6         try:
7             start = int(start)
8             except ValueError:
9                 return "Invalid start parameter; start must be an integer."
10    return flask.jsonify(d[start:])
```

EXERCISE 6

Add support for a `limit` parameter to the code you wrote for Exercise 5. The `limit` parameter should be optional. When passed with an integer value, the API should return no more than `limit` data points.

5.3.7 Additional Resources

- Flask JSON support
- Flask query parameter support

5.4 Containerizing Flask

As we have discussed previously, Docker containers are critical to packaging an application along with all of its dependencies, isolating it from other applications and services, and deploying it in a consistent and reproducible way across different platforms.

Here, we will walk through the process of containerizing a Flask application with Docker, and then using `curl` to interact with it as a containerized microservice. After going through this module, students should be able to:

- Assemble the different components needed for a containerized microservice into one directory.
- Establish and document requirements (e.g. dependencies, Python packages) for the project.
- Build and run in the background a containerized Flask microservice.
- Map ports on the ISP server to ports inside a container, and use `curl` with the the correct ports to make requests to and generate responses from the microservice.

5.4.1 Organize Your App Directory

First, create a “web” directory, and change directories to it:

```
[isp02]$ mkdir web
[isp02]$ cd web
```

Then, create a new `app.py` (or copy an existing one) into this folder. It should have the following contents:

```
1 from flask import Flask
2
3 app = Flask(__name__)
4
5 @app.route('/', methods = ['GET'])
6 def hello_world():
7     return 'Hello, world!\n'
8
9 @app.route('/<name>', methods = ['GET'])
10 def hello_name(name):
11     return 'Hello, {}!\n'.format(name)
12
13 if __name__ == '__main__':
14     app.run(debug=True, host='0.0.0.0')
```

5.4.2 Specify Requirements

The Python package manager `pip` can utilize a text file for managing package dependencies of your application. It is standard practice to capture the required libraries and packages for a project in a file called `requirements.txt`. For our example here, create a file called `requirements.txt` and add the following line:

```
Flask==2.0.3
```

This indicates that our project requires the Flask package, version number 2.0.3. You can specify your requirements in more lenient ways – for example, we could have put `Flask>=2.0.3` to indicate that any version greater than or equal to 2.0.3 would work, or we could have even put `Flask` with no version indicating we don’t care what version of Flask is installed.

Note:

- Specifying a package, such as Flask as a dependency instructs pip to install Flask *and all of its dependencies*. Those dependencies could in turn have dependencies, etc., and pip will take care of installing all of those.
- Specifying the exact version improves the odds that your application will work correctly because the packages that get installed will be the versions you specified. Therefore, it is usually best to specify the exact version of the library your application requires.

5.4.3 Build a Docker Image

As we saw in a previous section, we write up the recipe for our application installation process in a Dockerfile. Create a file called `Dockerfile` for our Flask microservice and add the following lines:

```
FROM python:3.9

RUN mkdir /app
WORKDIR /app
COPY requirements.txt /app/requirements.txt
RUN pip install -r /app/requirements.txt
COPY . /app

ENTRYPOINT ["python"]
CMD ["app.py"]
```

Here we see usage of the Docker `ENTRYPOINT` and `RUN` instructions, which essentially specify a default command (`python app.py`) that should be run when an instance of this image is instantiated.

Note also that we copied the `requirements.txt` file before copying the full current working directory. Why did we do that?

The answer has to do with how Docker caches image layers. We could have written the following instead:

```
FROM python:3.9

RUN mkdir /app
WORKDIR /app
COPY . /app
RUN pip install -r /app/requirements.txt

ENTRYPOINT ["python"]
CMD ["app.py"]
```

The above is actually shorter; i.e., fewer lines of code in the Dockerfile.

However, with the above approach, Docker is going to re-run the command `pip install -r /app/requirements.txt` every time there is any change to the contents of the current working directory (i.e., any time we change our app code or any other files). This is not a big deal with a small `requirements.txt` file and only a few packages to install, but as the `requirements.txt` file gets bigger, the time to install all the packages can be significant.

As a general rule of thumb, put more expensive (in term of time) operations whose are less likely to change at the beginning of your `Dockerfile` to maximize the value of the Docker image layer cache.

Save the file and build the image with the following command:

```
[isp02]$ docker build -t <username>/flask-helloworld:latest .
```

5.4.4 Run a Docker Container

To create an instance of your image (a “container”), use the following command:

```
[isp02]$ docker run --name "give-your-container-a-name" -d -p <your port number>:5000
↪<username>/flask-helloworld:latest"
```

For example:

```
[isp02]$ docker run --name "joe-helloworld-flask-app" -d -p 5050:5000 jstubbs/flask-
↪helloworld:latest
```

The `-d` flag detaches your terminal from the running container - i.e. it runs the container in the background. The `-p` flag maps a port on the ISP server (5050, in the above case) to a port inside the container (5000, in the above case). In the above example, the Flask app was set up to use the default port inside the container (5000), and we can access that through our specified port on ISP (5050).

Check to see that things are up and running with:

```
[isp02]$ docker ps -a
```

The list should have a container with the name you gave it, an UP status, and the port mapping that you specified.

If the above is not found in the list of running containers, try to debug with the following:

```
[isp02]$ docker logs "your-container-name"
-or-
[isp02]$ docker logs "your-container-number"
```

5.4.5 Access Your Microservice

Now for the payoff - you can use `curl` to interact with your Flask microservice by specifying the correct port on the ISP server. Following the example above, which was using port 5050:

```
[isp02]$ curl localhost:5050/
Hello, world!
[isp02]$ curl localhost:5050/Joe
Hello, Joe!
```

5.4.6 Clean Up

Finally, don't forget to stop your running container and remove it.

```
[isp02]$ docker ps -a | grep jstubbs
60be6788d73d    jstubbs/flask-helloworld:latest    "python app.py"    4 minutes ago    Up
↪4 minutes    0.0.0.0:5050->5000/tcp    joe-helloworld-flask-app
[isp02]$ docker stop 60be6788d73d
60be6788d73d
[isp02]$ docker rm 60be6788d73d
60be6788d73d
```

EXERCISE

Containerize your flask meteorite landings server from last week:

1. Create a Dockerfile for your server.
2. Build the image from the Dockerfile.
3. Run the server locally and test the /data endpoint using curl.

UNIT 6: INTRO TO DATABASES AND PERSISTENCE, CONTAINERIZING REDIS

In this unit, we will begin implementing a database using Redis to hold our data sets such that they persist beyond the lifetime of our application containers. We will also figure out how to query our data in Python by connecting our Flask API to our Redis database, and containerize both of our services.

6.1 Intro to Databases and Persistence

Application data that lives inside a container is ephemeral - it only persists for the lifetime of the container. We can use databases to extend the life of our application (or user) data, and even access it from outside the container.

After going through this module, students should be able to:

- Explain the differences between SQL and NoSQL databases
- Choose the appropriate type of database for a given application / data set
- Start and find the correct port for a Redis server
- Install and import the Redis Python library
- Add data to and retrieve data from a Redis database from a Python script

6.1.1 What's Our Motivation?

In this unit, we will begin the work to extend our Flask API to enable users to query and analyze data in our data sets. We want to expose this functionality through our Flask API, but there is an issue: the analysis to be performed may take “a long time” to compute, longer than the acceptable time window for an HTTP request/response cycle. We need a way to coordinate the work of computing the analysis in a separate Python program from our Flask API. The database will play a central role.

Our basic approach will be:

1. Our dataset will be stored in a database.
2. The user submits a request to a Flask endpoint describing some sort of analysis they wish to perform on the data.
3. We will create a separate Python program to perform the analysis. This program will retrieve the desired data from the database and store the final results in the database.
4. The Flask API will determine the status of the analysis by querying the database, and it will retrieve the final results of the analysis from the database to serve to the user when they are ready.

There are a lot of details to fill in over the course of the rest of the semester, but for now we are going to focus on getting data into and out of a database.

6.1.2 Intro to Databases

What is a database?

- A database is an organized collection of structured information, or data, typically stored electronically in a computer system.
- Databases provide a **query language** - a small, domain-specific language for interacting with the data. The query language is not like a typical programming language such as Python or C++; you cannot create large, complex programs with it. Instead, it is intended to allow for easy, efficient access to the data.

Why use a database?

- Our data needs permanence and we want to be able to stop and start our Flask API without losing data.
- We want multiple Python processes to be able to access the data at the same time, including Python processes that may be running on different computers.

Why not use a file?

- It is not easy to make a file accessible to Python processes on different computers / VMs.
- When multiple processes are reading from and writing to a file, race conditions can occur.
- With files, our Flask API would have to implement all the structure in the data.

Databases sometimes get classified into two broad categories: SQL databases (also called relational databases) and NoSQL databases.

6.1.3 SQL Databases

Structured Query Language (SQL) is a language for managing structured or relational data, where certain objects in the dataset are related to other objects in a formally or mathematically precise way. SQL is the language used when working with a relational database. You will often see SQL database technologies referred to as Relational Database Management Systems (RDBMS).

The SQL language is governed by an ISO standard, and relational databases are among the most popular databases in use today. SQL was originally based on a strong, theoretical framework called the Relational Model (and related concepts). However, today's SQL has departed significantly from that formal framework.

Popular open-source RDBMS include:

- MySQL
- Postgres
- Sqlite

6.1.4 NoSQL Databases

As the name implies, a NoSQL database is simply a database that does not use SQL. There are many different types of NoSQL databases, including:

- Time series databases
- Document stores
- Graph databases
- Simple key-value stores (like the one we will use in this class)

In some ways, it is easier to say what a NoSQL database isn't than what it is; some of the key attributes include:

- NoSQL databases do **NOT** use tables (data structured using rows and columns) connected through relations
- NoSQL databases store data in “collections”, “logical databases”, or similar containers
- NoSQL databases often allow for missing or different attributes on objects in the same collection
- Objects in one collection do not relate or link to objects in another collection
- For example, the objects themselves could be JSON objects without a pre-defined schema

SQL vs NoSQL

Comparing SQL and NoSQL is an apples to oranges comparison.

- Both SQL and NoSQL databases have advantages and disadvantages.
- The *primary* deciding factor should be the *shape* of the data and the requirements on the integrity of the data. In practice, many other considerations could come into play, such as what expertise the project team has.
- Also consider how the data may change over time, and how important is the relationship between the different types of data being stored.
- SQL databases “enforce” relationships between data types, including one-to-one, one-to-many, and many-to-many. When the integrity of the data is important, SQL databases are a good choice.
- In many NoSQL databases, the relationship enforcement must be programmed into the application. This can be error-prone and can increase the development effort needed to build the application. On the other hand, this can allow the database to be used for use cases where relationship enforcement is not possible.
- SQL databases historically cannot scale to the “largest” quantities of data because of the ACID (Atomicity, Consistency, Isolation, Durability) guarantees they make (though this is an active area of research).
- NoSQL databases trade ACID guarantees for weaker properties (e.g., “eventual consistency”) and greater scalability. It would be difficult to scale a relational database to contain the HTML of all websites on the internet or even all tweets ever published.

For the projects in this class, we are going to use Redis, a simple (NoSQL) “data structure” store. There are a few reasons for this choice:

- We need a flexible data model, as the structure of the data we will store in the database will be changing significantly over the course of the semester.
- We need a tool that is quick to learn and simple to use. This is not a databases course, and learning the SQL language would take significantly more time than we can afford.
- Redis can also easily be used as a task queue, which we will make use of in the asynchronous programming unit.

6.1.5 Redis

Redis is a very popular NoSQL database and “data structure store” with lots of advanced features including:

Key-value Store

Redis provides key-value store functionality:

- The items stored in a Redis database are structured as `key:value` objects.
- The primary requirement is that the `key` be unique across the database.
- A single Redis server can support multiple databases, indexed by an integer.
- The data itself can be stored as JSON.

Notes about Keys

Redis keys have the following properties/requirements:

- Keys are often strings, but they can be any “binary sequence”.
- Long keys can lead to performance issues.
- A format such as `<object_type>:<object_id>` is a good practice.

Notes on Values

- Values are typed; some of the primary types include:
 - Binary-safe strings
 - Lists (sorted collections of strings)
 - Sets (unsorted, unique collections of strings)
 - Hashes (maps of fields with associated values; both field and value are type `string`)
- There is no native “JSON” type; to store JSON, one can use an encoding and store the data as a binary-safe string, or one can use a hash and convert the object into and out of JSON.
- The basic string type is a “binary-safe” string, meaning it must include an encoding.
 - In Python terms, the string is stored and returned as type `bytes`.
 - By default, the string will be encoded with UTF-8, but we can specify the encoding when storing the string.
 - Since bytes are returned, it will be our responsibility to decode using the same encoding.

Hash Maps

- Hashes provide another way of storing dictionary-like data in Redis
- The values of the keys are type `string`

6.1.6 Running Redis

To use Redis on the class VM (ISP), we must have an instance of the Redis server running. For demonstration purposes, we will all share the same instance of Redis server on the same port (6379) running in a docker container.

Note: Please **do not** run the following command on your own. We only want to run one Redis container for the whole class at this time. We are including it here for documentation purposes only.

```
# start the Redis server on the command line:
[isp02]$ docker run -p 6379:6379 redis:6
1:C 18 Mar 2022 22:39:52.645 # oO00oO00oO00o Redis is starting oO00oO00oO00o
1:C 18 Mar 2022 22:39:52.645 # Redis version=6.2.6, bits=64, commit=00000000,
↳modified=0, pid=1, just started
1:C 18 Mar 2022 22:39:52.645 # Warning: no config file specified, using the default
↳config. In order to specify a config file use redis-server /path/to/redis.conf
1:M 18 Mar 2022 22:39:52.645 * monotonic clock: POSIX clock_gettime
1:M 18 Mar 2022 22:39:52.647 * Running mode=standalone, port=6379.
```

The Redis server is up and available. Although we could use the Redis CLI to interact with the server directly, in this class we will focus on the Redis Python library so we can interact with the server from our Python scripts.

Note: According to the log above, Redis is listening on the default port, **6379**.

First install the Redis Python library in your user account:

```
[isp02]$ pip3 install --user redis
```

Then open up an interactive Python interpreter to connect to the server:

```
[isp02]$ python3
Python 3.6.8 (default, Aug 7 2019, 17:28:10)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-39)] on linux
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>> import redis
>>>
>>> rd=redis.Redis(host='127.0.0.1', port=6379, db=<some integer>)
>>>
>>> type(rd)
<class 'redis.client.Redis'>
```

You've just created a Python client object to the Redis server called `rd`. This object has methods for adding, modifying, deleting, and analyzing data in the database instance, among other things.

Some quick notes:

- We are using the IP of the gateway (127.0.0.1) on our localhost and the default Redis port (6379).
- Redis organizes collections into “databases” identified by an integer index. Here, we are specifying `db=<some integer>`; if that database does not exist it will be created for us.

Note: Since we are sharing a single Redis server for this class, please use the integer associated with the last one or two digits of your Flask port; i.e., if your Flask port is 5001, use `db=1`, if your Flask port is 5023 use `db=23`. This will

ensure we do not collide with each other.

6.1.7 Working with Redis

We can create new entries in the database using the `.set()` method. Remember, entries in a Redis database take the form of a key:value pair. For example:

```
>>> rd.set('my_key', 'my_value')
True
```

This operation saved a key in the Redis server (`db=0`) called `my_key` and with value `my_value`. Note the method returned `True`, indicating that the request was successful.

We can retrieve it using the `.get()` method:

```
>>> rd.get('my_key')
b'my_value'
```

Note that `b'my_value'` was returned; in particular, Redis returned binary data (i.e., type `bytes`). The string was encoded for us (in this case, using Unicode). We could have been explicit and set the encoding ourselves. The `bytes` class has a `.decode()` method that can convert this back to a normal string, e.g.:

```
>>> rd.get('my_key')
b'my_value'
>>> type(rd.get('my_key'))
<class 'bytes'>
>>>
>>> rd.get('my_key').decode('utf-8')
'my_value'
>>> type( rd.get('my_key').decode('utf-8') )
<class 'str'>
```

6.1.8 Redis and JSON

A lot of the information we exchange comes in JSON or Python dictionary format. To store pure JSON as a binary-safe string value in a Redis database, we need to be sure to dump it as a string (`json.dumps()`):

```
>>> import json
>>> d = {'a': 1, 'b': 2, 'c': 3}
>>> rd.set('k1', json.dumps(d))
True
```

Retrieve the data again and get it back into JSON / Python dictionary format using the `json.loads()` method:

```
>>> rd.get('k1')
b'{"a": 1, "b": 2, "c": 3}'
>>> type(rd.get('k1'))
<class 'bytes'>
>>>
>>> json.loads(rd.get('k1'))
{'a': 1, 'b': 2, 'c': 3}
```

(continues on next page)

(continued from previous page)

```
>>> type(json.loads(rd.get('k1')))  
<class 'dict'>
```

Note: In some versions of Python, you may need to specify the encoding as we did earlier, e.g.:

```
>>> json.loads(rd.get('k1').decode('utf-8'))  
{ 'a': 1, 'b': 2, 'c': 3 }
```

Hashes

Hashes provide another way of storing dictionary-like data in Redis.

- Hashes are useful when different fields are encoded in different ways; for example, a mix of binary and unicode data.
- Each field in a hash can be treated with a separate decoding scheme, or not decoded at all.
- Use `hset()` to set a single field value in a hash or to set multiple fields at once.
- Use `hget()` to get a single field within a hash or to get all of the fields.

```
# set multiple fields on a hash  
>>> rd.hset('k2', mapping={'name': 'Joe', 'email': 'jstubbs@tacc.utexas.edu'})  
  
# set a single field on a hash  
>>> rd.hset('k2', 'type', 'instructor')  
  
# get one field  
>>> rd.hget('k2', 'name')  
b'Joe'  
  
# get all the fields in the hash  
>>> rd.hgetall('k2')  
{b'name': b'Joe', b'email': b'jstubbs@tacc.utexas.edu', b'type': b'instructor'}
```

Tip: You can use `rd.keys()` to return all keys from a database, and `rd.hkeys(key)` to return the list of keys within hash 'key', e.g.:

```
>>> rd.hkeys('k2')  
[b'name', b'email', b'type']
```

Exercise 1

Save the Meteorite Landings data (i.e., the `Meteorite_Landings.json` file from Unit 4) into Redis. Each landing data point should be stored as a single Redis object. Think about what data type you want to use in Redis for storing the data.

If needed, you can download the JSON file with the following command:

```
$ wget https://raw.githubusercontent.com/tacc/coe-332-sp22/main/docs/unit04/scripts/  
↪ Meteorite_Landings.json
```

Exercise 2

Check that you stored the data correctly:

- Check the total number of keys in your Redis database against the total number of objects in the JSON file.
- Read all of the landing objects out of Redis and check that each object has the correct fields.

6.1.9 Additional Resources

- [Redis Docs](#)
- [Redis Python Library](#)

6.2 Flask and Redis

Up to this point, we have been interacting with a shared Redis database instance running on the class ISP server. Here, we will each work with our own containerized instance of Redis, figure out how to interact with it by forwarding the port, update the command to mount a host directory to persist data, and connect it to our Flask app. After going through this module, students should be able to:

- Start a Redis container, connecting the appropriate inside port to a port on isp02.
- Mount a host volume on isp02 inside the Redis container to save data between restarts of Redis itself.
- Connect to the container from within a Flask application.

6.2.1 Start a Redis Container

Docker Hub has a wealth of official, public images. It is a good idea to pull the existing Redis image rather than build it ourselves, because it has all of the functionality we need as a base image.

Pull the official Redis image for version 6:

```
[isp02]$ docker pull redis:6  
6: Pulling from library/redis  
ae13dd578326: Pull complete  
e6f25d21ebb3: Pull complete  
601cc6106ba1: Pull complete  
5b8be2fd806e: Pull complete  
950c3791111a: Pull complete  
567b7ad78092: Pull complete
```

(continues on next page)

(continued from previous page)

```
Digest: sha256:81b50efa808d72f7965d4deecea18e42cab2fec25fafca447eb4bda615b9c8e4
Status: Downloaded newer image for redis:6
docker.io/library/redis:6
```

Start the Redis server in a container:

```
[isp02]$ docker run -p <your-redis-port>:6379 --name=<your_name>-redis redis:6
1:C 31 Mar 2021 16:48:11.939 # o000o000o000o Redis is starting o000o000o000o
...
1:M 31 Mar 2021 16:48:11.972 * Ready to accept connections
```

The above command will start the Redis container in the foreground which can be helpful for debugging, seeing logs, etc. However, you will have the need to start it in the background (detached or daemon mode). You can do that by adding the `-d` flag:

```
[isp02]$ docker run -d -p <your-redis-port>:6379 --name=<your_name>-redis redis:6
3a28cb265d5e09747c64a87f904f8184bd8105270b8a765e1e82f0fe0db82a9e
```

At this point, Redis is running and available from the isp02 host, but let's explore the persistence of the data in Redis.

We can open a Python shell, connect to our Redis container, and add some data:

```
>>> import redis
>>> rd = redis.Redis(host='127.0.0.1', port=<your_redis_port>, db=0)

# create some data
>>> rd.set('k', 'v')
True

# check that we can read the data
>>> rd.get('k')
b'v'
```

If we exit our Python shell and then go into a new Python shell and connect to Redis, what do we see?

```
>>> import redis
>>> rd = redis.Redis(host='127.0.0.1', port=<your_redis_port>, db=0)

# previous data is still there
>>> rd.get('k')
b'v'
```

Great! Redis has persisted our data across Python sessions. That was one of our goals. But what happens if we shut down the Redis container itself?

Let's find out by killing our Redis container and starting a new one.

```
# shut down the existing redis container using the name you gave it
[isp02]$ docker rm -f <your_name>-redis

# start a new redis container
[isp02]$ docker run -d -p <your-redis-port>:6379 --name=<your_name>-redis redis:6
```

Now go back into the Python shell and connect to Redis:

```
>>> import redis
>>> rd = redis.Redis(host='127.0.0.1', port=<your_redis_port>, db=0)

# previous data is gone!
>>> rd.get('k')

# no keys at all!
>>> rd.keys()
[]
```

Oops! All the data that was in Redis is gone. The problem is we are not permanently persisting the Redis data across different Redis containers. But wasn't that the whole point of using a database? Are we just back to where we started?

Actually, we only need two small changes to the way we are running the Redis container to make the Redis data persist across container executions.

6.2.2 Container Bind Mounts

A container bind mount (or just “mount” for short) is a way of replacing a file or directory in a container image with a file or directory on the host file system in a running container.

Bind mounts are specified with the `-v` flag to the `docker run` statement. The full syntax is

```
[isp02]$ docker run -v <host_path>:<container_path>:<mode> ...*additional docker run_
↪args*...
```

where:

- `<host_path>` and `<container_path>` are absolute paths in the host (respectively, container) file system and
- `<mode>` can take the value of `ro` for a read-only mount and `rw` for a read-write mount.

Note that `mode` is optional and defaults to read-write.

It is important to keep the following in mind when using bind mounts:

- If the container image originally contained a file or directory at the `<container_path>` these will be replaced entirely by the contents of `<host_path>`.
- If the container image did not contain contents at `<container_path>` the mount will still succeed and simply create a new file/directory at the path.
- If the `<mode>` is read-write (the default), any changes made by the running container will be reflected on the host file system. Note that the process running in the container still must have permission to write to the path.
- If `<host_path>` does not exist on the host, Docker will create a **directory** at the path and mount it into the container. **This may or may not be what you want.**

6.2.3 Persisting Data Across Redis Containers

We can use bind mounts to persist Redis data across container executions: the key point is that Redis can be started in a mode so that it periodically writes all of its data to the host.

From the Redis documentation, we see that we need to set the `--save` flag when starting Redis so that it writes its dataset to the file system periodically. The full syntax is:

```
--save <frequency> <number_of_backups>
```

where `<frequency>` is an integer, in seconds. We'll instruct Redis to write its data to the file system every second, and we'll keep just one backup.

Entrypoints and Commands in Docker Containers

Let's take a moment to revisit the difference between an entrypoint and a command in a Docker container image. When executing a container from an image, Docker uses both an `entrypoint` and an (optional) `command` to start the container. It combines the two using concatenation, with `entrypoint` first, followed by `command`.

When we use `docker run` to create and start a container from an existing image, we can choose to override either the command or the entrypoint that may have been specified in the image. Any string `<string>` passed after the `<image>` in the statement:

```
[isp02]$ docker run <options> <image> <string>
```

will override the `command` specified in the image, but the original `entrypoint` set for the image will still be used.

A common pattern when building Docker images is to set the `entrypoint` to the primary program, and set the `command` to a default set of options or parameters to the program.

Consider the following simple example:

```
# Dockerfile
FROM ubuntu
ENTRYPOINT ["ls"]
CMD ["-l"]
```

If I built and tagged this image as `jstubbs/ls`, then

```
# run with the default command, equivalent to "ls -l"
[isp02]$ docker run --rm -it jstubbs/ls
total 48
lrwxrwxrwx   1 root root    7 Jan  5 16:47 bin -> usr/bin
drwxr-xr-x   2 root root 4096 Apr 15  2020 boot
drwxr-xr-x   5 root root  360 Mar 23 18:37 dev
drwxr-xr-x   1 root root 4096 Mar 23 18:37 etc
drwxr-xr-x   2 root root 4096 Apr 15  2020 home
. . .

# override the command, but keep the entrypoint; equivalent to running "ls -a" (note the lack of "-l")
[isp02]$ docker run --rm -it jstubbs/ls -a
.  .dockerenv      boot  etc  lib   lib64  media  opt   root  sbin  sys  usr
.. bin             dev   home lib32 libx32 mnt   proc  run  srv  tmp  var
```

(continues on next page)

(continued from previous page)

```
# override the command, specifying a different directory
[isp02]$ docker run --rm -it jstubbbs/ls /root -la
total 16
drwx----- 2 root root 4096 Jan  5 16:50 .
drwxr-xr-x  1 root root 4096 Mar 23 18:38 ..
-rw-r--r--  1 root root 3106 Dec  5  2019 .bashrc
-rw-r--r--  1 root root  161 Dec  5  2019 .profile
```

Modifying the Command in the Redis Container

The official [redis](#) container image provides an entrypoint which starts the redis server (check out the [Dockerfile](#) if you are interested.).

Since the `save` option is a parameter, we can set it when running the redis server container by simply appending it to the end of the `docker run` command; that is,

```
[isp02]$ docker run <options> redis:6 --save <options>
```

Combining --save and Mounts for a Complete Solution

With `save`, we can instruct Redis to write the data to the file system, but we still need to save the files across container executions. That's where the bind mount comes in. But how do we know which directory to mount into Redis? Fortunately, the Redis documentation tells us what we need to know: Redis writes data to the `/data` directory in the container.

Putting all of this together, we can update the way we run our Redis container as follows:

```
[isp02]$ docker run -d -p <your-redis-port>:6379 -v </path/on/host>:/data --name=<your_
↪name>-redis redis:6 --save 1 1
```

Tip: You can use the `$(pwd)` shortcut for the present working directory.

For example, I might use:

```
[isp02]$ docker run -d -p 6379:6379 -v $(pwd)/data:/data:rw --name=joe-redis redis:6 --
↪save 1 1
```

Now, Redis should periodically write all of its state to the data directory. You should see a file called `dump.rdb` in the directory because we are using the default persistence mechanism for Redis. This will suffice for our purposes, but Redis has other options for persistence which you can read about [here](#) if interested.

Exercise 1

Test out persistence of your Redis data across Redis container restarts by starting a new Redis container using the method above, saving some data to it in a Python shell, shutting down the Redis container and starting a new one, and verifying back in the Python shell that the original data is still there.

6.2.4 Using Redis in Flask

Using Redis in our Flask apps is identical to using it in the Python shells that we have been using to explore with. We simply create a Python Redis client object using the `redis.Redis()` constructor. Since we might want to use Redis from different parts of the code, we'll create a function for generating the client:

```
1 def get_redis_client():
2     return redis.Redis(host='127.0.0.1', port=<your_port>, db=0)
```

Exercise 2

Note: This exercise will be part of the next home work assignment.

In the last module, we wrote a program to read the Meteorite Landings data (i.e., the `Meteorite_Landings.json` file from Unit 4) into Redis. In this exercise, let's turn this program into a Flask API with one route that handles both a POST and a GET.

- Use `/data` as the URL path for the one route.
- A POST request to `/data` should load the Meteorite Landings data into Redis.
- A GET request to `/data` should read the data out of Redis and return it as a JSON list.

For bonus points, implement an optional `start` query parameter that takes an integer and returns the Meteorite Landing data starting at the `start` index. Make sure to handle the case where `start` is provided but is not an integer!

UNIT 7: CONTAINER ORCHESTRATION

In this unit, we begin our study of container orchestration and the Kubernetes (“k8s”) system. We have already built a small HTTP application in the REST architecture using the flask framework. This HTTP application makes use of a database to persist state. In the coming weeks, we will add more components to our application, and this is very typical of a modern distributed system. As the number of components grows, the work required to deploy and maintain this system increases. Container orchestration systems such as k8s aid us in this deployment and management effort by allowing us to run our applications across a cluster of machines and use APIs to make changes to the application deployment over time.

At the end of this unit you will:

- Understand container orchestration and the basic Kubernetes architecture.
- Understand fundamental Kubernetes abstractions, including: `pod`, `deployment`, `persistent volume`, and `service`.
- Write a set of scripts to deploy your flask application to a Kubernetes cluster in your own private namespace.

7.1 Orchestration Overview

In this module, we introduce the basic notion of container orchestration. After going through this module, the student should be able to:

- Explain the challenges associated with deploying and managing distributed systems that container orchestration systems are designed to help with.
- Describe at a high level a few of the most common container orchestration systems.

A typical distributed system is comprised of multiple components. You have already developed a simple HTTP application that includes a Python program (using the flask library) and a database. In the subsequent weeks, we will add additional components.

Container orchestration involves deploying and managing containerized applications on one or more computers. There are a number of challenges involved in deploying and managing a distributed application, including:

- **Container execution and lifecycle management** – To run our application, we must not only start the containers initially but also manage the entire lifecycle of the container, including handling situations where containers are stopped and/or crash. We need to ensure the correct container image is used each time.
- **Configuration** – Most nontrivial applications involve some configuration. We must be able to provide configuration to our application’s components.
- **Networking** – The components in our distributed application will need to communicate with each other, and that communication will take place over a network.

- Storage – Some components, such as databases, will require access to persistent storage to save data to disc so that they can be restarted without information loss.

The above list is just an initial set of concerns when deploying distributed, containerized applications. As the size and number of components grows, some systems may encounter additional challenges, such as:

- Scaling – We may need to start up additional containers for one or more components to handle spikes in load on the system, and shut down these additional containers to save resources when the usage spike subsides.
- CPU and Memory management – The computers we run on have a fixed amount of CPU and memory, and in some cases, it can be important to ensure that no one container uses too many resources.
- Software version updates – How do we go from version 1 to version 2 of our software? We may have to update several components (or all of them) at once. What if there are multiple containers running for a given component? As we are performing the upgrade, is the system offline or can users still use it?

7.1.1 Orchestration Systems

Orchestration systems are built to help with one or more of the above challenges. Below we briefly cover some of the more popular, open-source container orchestration systems. This is by no means an exhaustive list.

Docker Compose

You have already seen a basic container orchestration system – Docker Compose. The Docker Compose system allows users to describe their application in a `docker-compose.yml` file, including the services, networks, volumes and other aspects of the deployment. Docker Compose:

- Runs Docker containers on a single computer, the machine where docker-compose is installed and run.
- Utilizes the Docker daemon to start and stop containers defined in the `docker-compose.yml` file.
- Also capable of creating volumes, networks, port bindings, and other objects available in the Docker API.

Docker compose is a great utility for single-machine deployments and, in particular, is a great system for “local development environments” where a developer has code running on her or his laptop.

Docker Swarm

Docker Swarm provides a container orchestration system to deploy applications across multiple machines running the Docker daemon. Docker Swarm works by creating a cluster (also known as a “swarm”) of computers and coordinating the starting and stopping of containers across the cluster. Docker Swarm:

- Runs Docker containers across a cluster of machines (a “swarm”), each running Docker.
- Coordinates container execution across the cluster.
- Similar API to Docker Compose: capable of creating networks spanning multiple computers as well as port-bindings, volumes, etc.

Mesos

Apache Mesos is a general-purpose cluster management system for deploying both containerized and non-containerized applications across multiple computers. Mesos by itself is quite low-level and requires the use of *frameworks* to deploy actual applications. For example, Marathon is a popular Mesos framework for deploying containerized applications, while the Mesos Hydra framework can be used for deploying MPI-powered applications, such as those used in traditional HPC applications.

Kubernetes

Kubernetes (often abbreviated as “k8s”) is a container orchestration system supporting Docker as well other container runtimes that conform to the Container Runtime Interface (CRI) such as containerd and cri-o. While Kubernetes focuses entirely on containerized applications (unlike Mesos) and is not as similar to Docker Compose as Docker Swarm is, it provides a number of powerful features for modern, distributed systems management. Additionally, Kubernetes is available as a service on a large number of commercial cloud providers, including Amazon, Digital Ocean, Google, IBM, Microsoft, and many more, and TACC provides multiple Kubernetes clusters in support of various research projects.

- Supports running containerized applications across a cluster of machines for container runtimes conforming to the Container Runtime Interface (CRI), including Docker.
- Provides powerful features for managing distributed applications.
- Available as a service from TACC and a number of commercial cloud providers.

7.1.2 Additional Resources

- [Docker Compose Reference](#)
- [Docker Swarm](#)
- [Apache Mesos Documentation](#)
- [Marathon](#)
- [Mesos Hydra](#)
- [Kubernetes Documentation](#)

7.2 Kubernetes - Overview and Introduction to Pods

In this section we give an overview of the Kubernetes system and introduce the first major Kubernetes abstraction, the Pod.

After going through this module, the student should be able to:

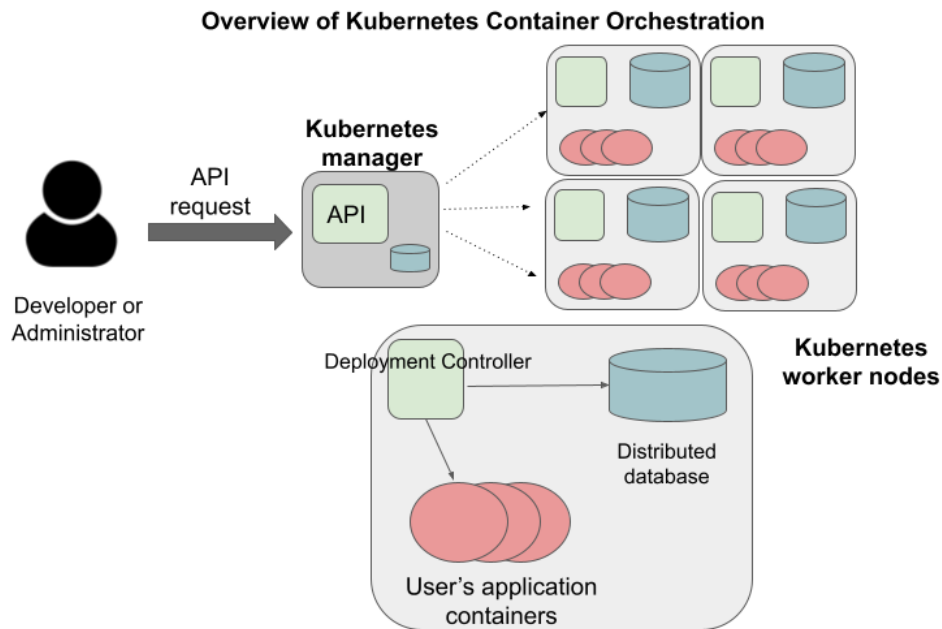
- Connect to the class Kubernetes cluster and issue basic commands using `kubectl`.
- Describe a Kubernetes pod in a yaml file and use `kubectl` to deploy the pod to the cluster.
- Retrieve details about the pod including status and logs.
- Use labels to organize pods deployed to the Kubernetes cluster.

7.2.1 Kubernetes Overview

Kubernetes (k8s) is itself a distributed system of software components that run a cluster of one or more machines (physical computers or virtual machines). Each machine in a k8s cluster is either a “manager” or a “worker” node.

Users communicate with k8s by making requests to its API. The following steps outline how Kubernetes works at a high level:

- 1) Requests to k8s API describe the user’s *desired state* on the cluster; for example, the desire that 3 containers of a certain image are running.
- 2) The k8s API schedules new containers to run on one or more worker nodes.
- 3) After the containers are started, the Kubernetes deployment controller, installed on each worker node, monitors the containers on the node.
- 4) The k8s components, including the API and the deployment controllers, maintain both the *desired state* and the *actual state* in a distributed database. The components continuously coordinate together to make the actual state converge to the desired state.



Note: It is important to note that most of the time, the k8s API as well as the worker nodes are running on separate machines from the machine we use to interact with k8s (i.e., make API requests to it). The machine we use to interact with k8s only needs to have the k8s client tools installed, and in fact, as the k9s API is available over HTTP, we don’t strictly speaking require the tools – we could use curl or some other http client – but the tools make interacting with the API much easier.

Connecting to the TACC Kubernetes Instance

In this class, we will use a Kubernetes cluster running at TACC that we have created for use in this class for deploying our applications. To simplify the process of accessing the Kubernetes cluster, we have enabled connectivity to it from the isp02 host. Therefore, any time you want to work with k8s, simply SSH to isp02 with your TACC username as you have throughout the semester and then ssh to coe332-k8s.tacc.cloud:

```
[laptop] $ ssh <tacc_username>@isp02.tacc.utexas.edu  
[isp02] $ ssh <tacc_username>@coe332-k8s.tacc.cloud
```

Note: The COE 332 kubernetes cluster is not available on the public internet for security reasons. You must first SSH to isp02 before ssh'ing to coe332-k8s.tacc.cloud.

First Commands with k8s

We will use the Kubernetes Command Line Interface (CLI) referred to as “kubectl” (pronounced “Kube control”) to make requests to the Kubernetes API. We could use any HTTP client, including a command-line client such as curl, but kubectl simplifies the process of formatting requests.

The kubectl software should already be installed and configured to use the Freetail K8s cluster. Let's verify that is the case by running the following:

```
$ kubectl version -o yaml
```

You should see output similar to the following:

```
clientVersion:  
  buildDate: "2022-03-16T15:58:47Z"  
  compiler: gc  
  gitCommit: c285e781331a3785a7f436042c65c5641ce8a9e9  
  gitTreeState: clean  
  gitVersion: v1.23.5  
  goVersion: go1.17.8  
  major: "1"  
  minor: "23"  
  platform: linux/amd64  
  
serverVersion:  
  buildDate: "2022-02-16T12:32:02Z"  
  compiler: gc  
  gitCommit: e6c093d87ea4cbb530a7b2ae91e54c0842d8308a  
  gitTreeState: clean  
  gitVersion: v1.23.4  
  goVersion: go1.17.7  
  major: "1"  
  minor: "23"  
  platform: linux/amd64
```

This command made an API request to the TACC k8s cluster and returned information about the version of k8s running there (under serverVersion) as well as the version of the kubectl that we are running (under clientVersion).

Note: The output of the `kubectl` command was `yaml` because we used the `-o yaml` flag. We could have asked for the output to be formatted in `json` with `-o json`. The `-o` flag is widely available on `kubectl` commands.

7.2.2 Introduction to Pods

Pods are a fundamental abstraction within Kubernetes and are the most basic unit of computing that can be deployed onto the cluster. A pod can be thought of as generalizing the notion of a container: a pod contains one or more containers that are tightly coupled and need to be scheduled together, on the same computer, with access to a shared file system and a shared network address.

Note: By far, the majority pods you will meet in the wild, including the ones used in this course, will only include one container. A pod with multiple containers can be thought of as an “advanced” use case.

7.2.3 Hello, Kubernetes

To begin, we will define a pod with one container. As we will do with all the resources we want to create in k8s, we will describe our pod in a `yaml` file.

Create a file called `pod-basic.yaml`, open it up in an editor and paste the following code in:

```
---
apiVersion: v1
kind: Pod
metadata:
  name: hello
spec:
  containers:
  - name: hello
    image: ubuntu:18.04
    command: ['sh', '-c', 'echo "Hello, Kubernetes!" && sleep 3600']
```

Let’s break this down. The top four attributes are common to all k8s resource descriptions:

- `apiVersion` – describes what version of the k8s API we are working in. We are using `v1`.
- `kind` – tells k8s what kind of resource we are describing, in this case a `Pod`.
- `metadata` – in general, this is additional information about the resource we are describing that doesn’t pertain to its operation. Here, we are giving our pod a `name`, `hello`.
- `spec` – This is where the actual description of the resource begins. The contents of this stanza vary depending on the `kind` of resource you are creating. We go into more details on this in the next section.

Warning: Only one Kubernetes object of a specific `kind` can have a given `name` at a time. If you define a second pod with the same `name` you will overwrite the first pod. This is true of all the different types of k8s objects we will be creating.

7.2.4 The Pod Spec

In k8s, you describe resources you want to create or update using a `spec`. The required and optional parameters available depend on the kind of resource you are describing.

The pod spec we defined looked like this:

```
spec:
  containers:
  - name: hello
    image: ubuntu:18.04
    command: ['sh', '-c', 'echo "Hello, Kubernetes!" && sleep 3600']
```

There is just one stanza, the `containers` stanza, which is a list of containers (recall that pods can contain multiple containers). Here we are defining just one container. For each container, we provide:

- `name` (optional) – this is the name of the container, similar to the name attribute in Docker.
- `image` (required) – the image we want to use for the container, just like with Docker.
- `command` (optional) – the command we want to run in the container. Here we are running a short BASH script.

7.2.5 Creating the Pod In K8s

We are now ready to create our pod in k8s. To do so, we use the `kubectl apply` command. In general, when you have a description of a resource that you want to create or update in k8s, the `kubectl apply` command can be used.

In this case, our description is contained in a file, so we use the `-f` flag. Try this now:

```
$ kubectl apply -f pod-basic.yml
```

If all went well and k8s accepted your request, you should see an output like this:

```
pod/hello created
```

Note: The message `pod/hello created` indicates that the description of the pod was valid, that k8s has saved the pod definition in its database and that it is working on starting the pod on the cluster. It does **not** mean the pod is already created/running on the cluster.

In practice, we won't be creating many Pod resources directly – we'll be creating other resources, such as `deployments` that are made up of pods – but it is important to understand pods and to be able to work with pods using `kubectl` for debugging and other management tasks.

Note: The pod we just created is running on the k8s cluster, NOT on `isp02` and NOT on `coe332-k8s.tacc.cloud`. You will not be able to find it using commands like `docker ps`, etc.

During the lecture, we'll draw a picture here to help explain what is going on.

7.2.6 Working With Pods

We can use additional `kubectl` commands to get information about the pods we run on k8s.

Listing Pods

For example, we can list the pods on the cluster with `kubectl get <object_type>` – in this case, the object type is “pods”:

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
ds-staging-6df657fbd-tbht5	1/1	Running	0	34d
elasticsearch-748f666f4f-svf2h	1/1	Running	0	76d
hello	1/1	Running	0	3s
kibana-f5b79569f-r4pl6	1/1	Running	0	78d
sidecartest-5454b7d49b-q8fvw	3/3	Running	472	78d

The output is fairly self-explanatory. We see a line for every pod which includes its name, status, the number of times it has been restarted and its age. Our `hello` pod is listed above, with an age of 3s because we just started it but it is already RUNNING. Several additional pods are listed in my output above due to prior work sessions.

A Word on Authentication and Namespaces

With all the students running their own pods on the same k8s cluster, you might be wondering why you only see your pod or why you don't see my pods? The reason is that when you make an API request to k8s, you tell the API who you are and what *namespace* you want to make the request in. Namespaces in k8s are logically isolated views or partitions of the k8s objects. Your `kubectl` client is configured to make requests in a namespace that is private to you; we set these namespaces up for COE 332.

We set up the k8s client configuration ahead of time for you. The client configuration resides in the file `~/.kube/config`. Take a look at the file if you are interested.

Getting and Describing Pods

We can pass the pod name to the `get` command – i.e., `kubectl get pods <pod_name>` – to just get information on a single pod

```
$ kubectl get pods hello
```

NAME	READY	STATUS	RESTARTS	AGE
hello	1/1	Running	0	3m1s

The `-o wide` flag can be used to get more information:

```
$ kubectl get pods hello -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE	NOMINATED
hello	1/1	Running	0	14m	172.16.178.1	kube-3.novalocal	<none>

Finally, the `kubectl describe <resource_type> <resource_name>` command gives additional information, including the k8s events at the bottom. While we won't go into the details now, this information can be helpful when troubleshooting a pod that has failed:

```

$ kubectl describe pods hello
Name:          hello
Namespace:     jstubbs
Priority:       0
Node:          kube-3.novalocal/10.0.2.10
Start Time:    Fri, 25 Mar 2022 21:52:11 +0000
Labels:        <none>
Annotations:   cni.projectcalico.org/containerID: ↵
               ↵ cfb4e44a8f265f9cc2bde568606d2c01f37a72b238fa0cbdbd2478c9008c2dc0
               cni.projectcalico.org/podIP: 172.16.178.1/32
               cni.projectcalico.org/podIPs: 172.16.178.1/32
Status:        Running
IP:            172.16.178.1
IPs:           IP: 172.16.178.1
Containers:
  hello:
    Container ID:  containerd://
    ↵ 6b06364267fcae79afbb132969608e26ab4e97473076f5e7856196bb9f88f44f
    Image:         ubuntu:18.04
    Image ID:      docker.io/library/
    ↵ ubuntu@sha256:d8ac28b7bec51664c6b71a9dd1d8f788127ff310b8af30820560973bcfc605a0
    Port:          <none>
    Host Port:     <none>
    Command:
      sh
      -c
      echo "Hello, Kubernetes!" && sleep 3600
  State:          Running
    Started:      Fri, 25 Mar 2022 21:52:20 +0000
  Ready:         True
  Restart Count:  0
  Environment:   <none>
  Mounts:
    /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-n7j9g (ro)
  Conditions:
    Type              Status
    Initialized       True
    Ready             True
    ContainersReady   True
    PodScheduled      True
  Volumes:
    kube-api-access-n7j9g:
      Type:          Projected (a volume that contains injected data from multiple ↵
    ↵ sources)
      TokenExpirationSeconds: 3607
      ConfigMapName:        kube-root-ca.crt
      ConfigMapOptional:    <nil>
      DownwardAPI:          true
      QoS Class:             BestEffort
      Node-Selectors:        <none>
      Tolerations:           node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
                           node.kubernetes.io/unreachable:NoExecute op=Exists for 300s

```

(continues on next page)

(continued from previous page)

Events:				
Type	Reason	Age	From	Message
----	-----	----	----	-----
Normal	Scheduled	10m	default-scheduler	Successfully assigned jstubbs/hello to ↵
↵ kube-3.novalocal				
Normal	Pulling	10m	kubelet	Pulling image "ubuntu:18.04"
Normal	Pulled	9m52s	kubelet	Successfully pulled image "ubuntu:18.04" ↵
↵ in 7.784328594s				
Normal	Created	9m52s	kubelet	Created container hello
Normal	Started	9m52s	kubelet	Started container hello

Getting Pod Logs

Finally, we can use `kubectl logs <pod_name>` command to get the logs associated with a pod:

```
$ kubectl logs hello
Hello, Kubernetes!
```

Note that the `logs` command does not include the resource name (“pods”) because it only can be applied to pods. The `logs` command in k8s is equivalent to that in Docker; it returns the standard output (stdout) of the container.

Using Labels

In the pod above we used the `metadata` stanza to give our pod a name. We can use `labels` to add additional metadata to a pod. A label in k8s is nothing more than a `name: value` pair that we create to organize objects in a meaningful way. We can choose any value for `name` and `value` that we wish but they must be strings. If you want to use a number like “10” for a label name or value, be sure to enclose it in quotes (i.e., “10”).

You can think of these `name:value` pairs as variables and values. So for example, you might create a label called `shape` with values `circle`, `triangle`, `square`, etc. A more realistic label might be `component_type` with values `api`, `database`, `worker`, etc. Multiple pods can have the same `name:value` label.

Let’s use the pod definition above to create a new pod with a label.

Create a file called `pod-label.yml`, open it up in an editor and paste the following code in:

```
---
apiVersion: v1
kind: Pod
metadata:
  name: hello-label
  labels:
    version: "1.0"
spec:
  containers:
    - name: hello
      image: ubuntu:18.04
      command: ['sh', '-c', 'echo "Hello, Kubernetes!" && sleep 3600']
```

Let’s create this pod using `kubectl apply`:

```
$ kubectl apply -f pod-label.yml
pod/hello-label created
```

Now when we list our pods, we should see it

```
$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
hello         1/1     Running   0           22m
hello-label   1/1     Running   0           22s
```

Filtering By Labels With Selectors

Labels are useful because we can use `selectors` to filter our results for a given label name and value. To specify a label name and value, use the following syntax: `--selector "<label_name>=<label_value>"`.

For instance, we can search for pods with the version 1.0 label like so:

```
$ kubectl get pods --selector "version=1.0"
NAME          READY   STATUS    RESTARTS   AGE
hello-label   1/1     Running   0           4m58s
```

We can also just use the label name to filter with the syntax `--selector "<label_name>"`. This will find any pods with the label `<label_name>`, regardless of the value.

7.2.7 Additional Resources

- [k8s Pod Reference](#)

7.3 Deployments

In this module, we discuss the Kubernetes abstraction called “deployments”. After working through this module, students should be able to:

- Explain the types of workloads that should be scheduled using the deployment abstraction in Kubernetes.
- Describe a deployment in a yaml file and schedule the deployment on a Kubernetes cluster using `kubectl`.
- Exec into a pod within a deployment and scale the pods associated with a deployment.
- Define an image naming and tagging scheme to manage the development and deployment lifecycle of an application.
- Utilize Kubernetes mounts, volumes and persistent volume claims to persist application data across pod/container restarts.

7.3.1 Introduction to Deployments

Deployments are an abstraction and resource type in Kubernetes that can be used to represent long-running application components, such as databases, REST APIs, web servers, or asynchronous worker programs. The key idea with deployments is that they should *always be running*.

Imagine a program that runs a web server for a blog site. The blog website should always be available, 24 hours a day, 7 days a week. If the blog web server program crashes, it would ideally be restarted immediately so that the blog site becomes available again. This is the main idea behind deployments.

Deployments are defined with a pod definition and a replication strategy, such as, “run 3 instances of this pod across the cluster” or “run an instance of this pod on every worker node in the k8s cluster.”

For this class, we will define deployments for our flask application and its associated components, as deployments come with a number of advantages over defining “raw” pods. Deployments:

- Can be used to run multiple instances of a pod, to allow for more computing to meet demands put on a system.
- Are actively monitored by k8s for health – if a pod in a deployment crashes or is otherwise deemed unhealthy, k8s will try to start a new one automatically.

7.3.2 Creating a Basic Deployment

We will use yaml to describe a deployment just like we did in the case of pods. Copy and paste the following into a file called `deployment-basic.yml`

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-deployment
  labels:
    app: hello-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hello-app
  template:
    metadata:
      labels:
        app: hello-app
    spec:
      containers:
        - name: hellos
          image: ubuntu:18.04
          command: ['sh', '-c', 'echo "Hello, Kubernetes!" && sleep 3600']
```

Let’s break this down. Recall that the top four attributes are common to all k8s resource descriptions, however it is worth noting:

- `apiVersion` – We need to use version `apps/v1` here. In k8s, different functionalities are packaged into different APIs. Deployments are part of the `apps/v1` API, so we must specify that here.
- `metadata` – The `metadata.name` gives our deployment object a name. This part is similar to when we defined pods. We are also using `labels`. Recall that k8s uses labels to allow objects to refer to other objects in a decoupled way. A label in k8s is nothing more than a `name: value` pair that users create to organize objects and add information meaningful to the user. In this case, `app` is the name and `hello-app` is the value. Conceptually, you can think of label names like variables and labels values as the value for the variable. In some other deployment, we may choose to use label `app: profiles` to indicate that the deployment is for the “profiles” app.

Let’s look at the `spec` stanza for the deployment above.

- `replicas` – Defines how many pods we want running at a time for this deployment, in this case, we are asking that just 1 pod be running at a time.
- `selector` – This is how we tell k8s where to find the pods to manage for the deployment. Note we are using labels again here, the `app: hello-app` label in particular.

- `template` – Deployments match one or more pod descriptions defined in the template. Note that in the metadata of the template, we provide the same label (`app: hello-app`) as we did in the `matchLabels` stanza of the selector. This tells k8s that this spec is part of the deployment.
- `template.spec` – This is a pod spec, just like we worked with last time.

Note: If the labels, selectors and `matchLabels` seems confusing and complicated, that's understandable. These semantics allow for complex deployments that dynamically match different pods, but for the deployments in this class, you will not need this extra complexity. As long as you ensure the label in the `template` is the same as the label in the `selector.matchLabels` your deployments will work. It's worth pointing out that the first use of the `app: hello-app` label for the deployment itself (lines 5 and 6 of the yaml) could be removed without impacting the end result.

We create a deployment in k8s using the `apply` command, just like when creating a pod:

```
$ kubectl apply -f deployment-basic.yml
```

If all went well, k8s response should look like:

```
deployment.apps/hello-deployment created
```

We can list deployments, just like we listed pods:

```
$ kubectl get deployments
NAME                READY    UP-TO-DATE    AVAILABLE    AGE
hello-deployment    1/1      1              1            1m
```

We can also list pods, and here we see that k8s has created a pod for our deployment for us:

```
$ kubectl get pods
NAME                                READY    STATUS    RESTARTS    AGE
hello                              1/1      Running   0            29m
hello-deployment-9794b4889-kms7p  1/1      Running   0            1m
```

Note that we see our “hello” pod from earlier as well as the pod “hello-deployment-9794b4889-kms7p” that k8s created for our deployment. We can use all the `kubectl` commands associated with pods, including listing, describing and getting the logs. In particular, the logs for our “hello-deployment-9794b4889-kms7p” pod prints the same “Hello, Kubernetes!” message, just as was the case with our first pod.

7.3.3 Deleting Pods

However, there is a fundamental difference between the “hello” pod we created before and our “hello” deployment which we have alluded to. This difference can be seen when we delete pods.

To delete a pod, we use the `kubectl delete pods <pod_name>` command. Let's first delete our hello deployment pod:

```
$ kubectl delete pods hello-deployment-9794b4889-kms7p
```

It might take a little while for the response to come back, but when it does you should see:

```
pod "hello-deployment-9794b4889-kms7p" deleted
```

If we then immediately list the pods, we see something interesting:

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
hello	1/1	Running	0	33m
hello-deployment-9794b4889-sx6jc	1/1	Running	0	9s

We see a new pod (in this case, “hello-deployment-9794b4889-sx6jc”) was created and started by k8s for our hello deployment automatically! k8s did this because we instructed it that we wanted 1 replica pod to be running in the deployment’s spec – this was the *desired* state – and when that didn’t match the actual state (0 pods) k8s worked to change it. Remember, deployments are for programs that should *always be running*.

What do you expect to happen if we delete the original “hello” pod? Will k8s start a new one? Let’s try it

```
$ kubectl delete pods hello
pod "hello" deleted

$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
hello-deployment-9794b4889-sx6jc	1/1	Running	0	4m

k8s did not start a new one. This “automatic self-healing” is one of the major difference between deployments and pods.

7.3.4 Scaling a Deployment

If we want to change the number of pods k8s runs for our deployment, we simply update the `replicas` attribute in our deployment file and apply the changes. Let’s modify our “hello” deployment to run 4 pods. Modify `deployment-basic.yml` as follows:

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-deployment
  labels:
    app: hello-app
spec:
  replicas: 4
  selector:
    matchLabels:
      app: hello-app
  template:
    metadata:
      labels:
        app: hello-app
    spec:
      containers:
        - name: hellos
          image: ubuntu:18.04
          command: ['sh', '-c', 'echo "Hello, Kubernetes!" && sleep 3600']
```

Apply the changes with:

```
$ kubectl apply -f deployment-basic.yml
deployment.apps/hello-deployment configured
```

When we list pods, we see k8s has quickly implemented our requested change:

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
hello-deployment-9794b4889-mk6qw	1/1	Running	0	11s
hello-deployment-9794b4889-sx6jc	1/1	Running	0	15m
hello-deployment-9794b4889-v2mb9	1/1	Running	0	11s
hello-deployment-9794b4889-vp6mp	1/1	Running	0	11s

7.3.5 EXERCISE

- 1) Delete several of the hello deployment pods and see what happens.
- 2) Scale the number of pods associated with the hello deployment back down to 1.

7.3.6 Updating Deployments with New Images

When we have made changes to the software or other aspects of a container image and we are ready to deploy the new version to k8s, we have to update the pods making up the corresponding deployment. We will use two different strategies, one for our “test” environment and one for “production”.

Test Environments

A standard practice in software engineering is to maintain one or more “pre-production” environments, often times called “test” or “quality assurance” environments. These environments look similar to the “real” production environment where actual users will interact with the software, but few if any real users have access to them. The idea is that software developers can deploy new changes to a test environment and see if they work without the risk of potentially breaking the software for real users if they encounter unexpected issues.

Test environments are essential to maintaining quality software, and every major software project the Cloud and Interactive Computing group at TACC develops makes use of multiple test environments. We will have you create separate test and production environments as part of building the final project in this class.

It is also common practice to deploy changes to the test environment often, as soon as code is ready and tests are passing on a developer’s laptop. We deploy changes to our test environments dozens of times a day while a large enterprise like Google may deploy many thousands of times a day. We will learn more about test environments and automated deployment strategies in the Continuous Integration section.

Image Management and Tagging

As you have seen, the tag associated with a Docker image is the string after the `:` in the name. For example, `ubuntu:18.04` has a tag of `18.04` representing the version of Ubuntu packaged in the image, while `jstubby/hello-flask:dev` has a tag of `dev`, in this case indicating that the image was built from the dev branch of the corresponding git repository. Use of tags should be deliberate and is an important detail in a well designed software development release cycle.

Once you have created a deployment for a pod with a given image, there are two basic approaches to deploying an updated version of the container images to k8s:

1. Use a new image tag or
2. Use the same image tag and instruct k8s to download the image again.

Using new tags is useful and important whenever you may want to be able to recover or revert back to the previous image, but on the other hand, it can be tedious to update the tag every time there is a minor change to a software image.

Therefore, we suggest the following guidelines for image tagging:

1. During development when rapidly iterating and making frequent deployments, use a tag such as `dev` to indicate the image represents a development version of the software (and is not suitable for production) and simply overwrite the image tag with new changes. Instruct k8s to always try to download a new version of this tag whenever it creates a pod for the given deployment (see next section).
2. Once the primary development has completed and the code is ready for end-to-end testing and evaluation, begin to use new tags for each change. These are sometimes called “release candidates” and therefore, a tagging scheme such as `rc1`, `rc2`, `rc3`, etc., can be used for tagging each release candidate.
3. Once testing has completed and the software is ready to be deployed to production, tag the image with the version of the software. There are a number of different schemes for versioning software, such as Semantic Versioning (<https://semver.org/>), which will discuss later in the semester, time permitting.

ImagePullPolicy

When defining a deployment, we can specify an `ImagePullPolicy` which instructs k8s about when and how to download the image associated with the pod definition. For our test environments, we will instruct k8s to always try and download a new version of the image whenever it creates a new pod. We do this by specifying `imagePullPolicy: Always` in our deployment.

For example, we can add `imagePullPolicy: Always` to our `hello-deployment` as follows:

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-deployment
  labels:
    app: hello-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hello-app
  template:
    metadata:
      labels:
        app: hello-app
    spec:
      containers:
        - name: hellos
          imagePullPolicy: Always
          image: ubuntu:18.04
          command: ['sh', '-c', 'echo "Hello, Kubernetes!" && sleep 3600']
```

and now k8s will always try to download the latest version of `ubuntu:18.04` from Docker Hub every time it creates a new pod for this deployment. As discussed above, using `imagePullPolicy: Always` is nice during active development because you ensure k8s is always deploying the latest version of your code. Other possible values include

IfNotPresent (the current default) which instructs k8s to only pull the image if it doesn't already exist on the worker node. This is the proper setting for a production deployment in most cases.

Deleting Pods to Update the Deployment

Note that if we have an update to our :dev image and we have set `imagePullPolicy: Always` on our deployment, all we have to do is delete the existing pods in the deployment to get the updated version deployed: as soon as we delete the pods, k8s will determine that an insufficient number of pods are running and try to start new ones. The `imagePullPolicy` instructs k8s to first try and download a newer version of the image.

7.3.7 Mounts, Volumes and Persistent Volume Claims

Some applications such as databases need access to storage where they can save data that will persist across container starts and stops. We saw how to solve this with Docker using a host bind mount. With k8s, the pods (containers) get started automatically for us on different nodes in the clusters, so a mount from a host won't work. Which host would we use to store the files to be persisted?

The solution in k8s involves a combination of what are called volume mounts, volumes and persistent volume claims. The basic idea is similar to that of a Docker host bind mount – we'll be replacing some location in the container image with some data stored outside of the container. But in order to handle the fact that the application container could get started on different compute nodes, we'll utilize a backend "storage resource" which provides block storage over a network.

Create a new file, `deployment-pvc.yml`, with the following contents, replacing "<username>" with your username:

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-pvc-deployment
  labels:
    app: hello-pvc-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hello-pvc-app
  template:
    metadata:
      labels:
        app: hello-pvc-app
    spec:
      containers:
        - name: hellos
          image: ubuntu:18.04
          command: ['sh', '-c', 'echo "Hello, Kubernetes!" >> /data/out.txt && sleep 3600
↪']
          volumeMounts:
            - name: hello-<username>-data
              mountPath: "/data"
      volumes:
        - name: hello-<username>-data
```

(continues on next page)

(continued from previous page)

```

persistentVolumeClaim:
  claimName: hello-<username>-data

```

Note: Be sure to replace `<username>` with your actual username in the YAML above.

We have added a `volumeMounts` stanza to `spec.containers` and we added a `volumes` stanza to the `spec`. These have the following effects:

- The `volumeMounts` include a `mountPath` attribute whose value should be the path in the container that is to be provided by a volume instead of what might possibly be contained in the image at that path. Whatever is provided by the volume will overwrite anything in the image at that location.
- The `volumes` stanza states that a volume with a given name should be fulfilled with a specific `persistentVolumeClaim`. Since the volume name (`hello-<username>-data`) matches the name in the `volumeMounts` stanza, this volume will be used for the `volumeMount`.
- In k8s, a persistent volume claim makes a request for some storage from a storage resource configured by the k8s administrator in advance. While complex, this system supports a variety of storage systems without requiring the application engineer to know details about the storage implementation.

Note also that we have changed the command to redirect the output of the `echo` command to the file `/data/out.txt`. This means that we should not expect to see the output in the logs for pod but instead in the file inside the container.

However, if we create this new deployment and then list pods we see something curious:

```

$ kubectl apply -f deployment-pvc.yml
$ kubectl get pods

```

NAME	READY	STATUS	RESTARTS	AGE
hello-deployment-9794b4889-mk6qw	1/1	Running	1	62m
hello-deployment-9794b4889-sx6jc	1/1	Running	1	78m
hello-deployment-9794b4889-v2mb9	1/1	Running	1	62m
hello-deployment-9794b4889-vp6mp	1/1	Running	1	62m
hello-pvc-deployment-74f985fffb-g9zd7	0/1	Pending	0	4m22s

Our “hello-deployment” pods are still running fine but our new “hello-pvc-deployment” pod is still in “Pending” status. It appears to be stuck. What could be wrong?

We can ask k8s to describe that pod to get more details:

```

$ kubectl describe pods hello-pvc-deployment-74f985fffb-g9zd7

```

```

Name:          hello-pvc-deployment-74f985fffb-g9zd7
Namespace:     designsafe-jupyter-stage
Priority:       0
Node:          <none>
Labels:        app=hello-pvc-app
               pod-template-hash=74f985fffb
<... some output omitted ...>
Tolerations:   node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
               node.kubernetes.io/unreachable:NoExecute op=Exists for 300s

Events:
  Type     Reason             Age   From                  Message
  ----     -
  Warning   FailedScheduling   63s   default-scheduler    0/1 nodes are available: 1

```

→ persistentvolumeclaim "hello-jstubby-data" not found.

(continues on next page)

(continued from previous page)

At the bottom we see the “Events” section contains a clue: persistentvolumeclaim “hello-jstubby-data” not found.

This is our problem. We told k8s to fill a volume with a persistent volume claim named “hello-jstubby-data” but we never created that persistent volume claim. Let’s do that now!

Open up a file called `hello-pvc.yml` and copy the following contents, being sure to replace `<username>` with your TACC username:

```
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: hello-<username>-data
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: nfs
  resources:
    requests:
      storage: 1Gi
```

Note: Again, be sure to replace `<username>` with your actual username in the YAML above.

We will use this file to create a persistent volume claim against the storage that has been set up in the TACC k8s cluster. In order to use this storage, you do need to know the storage class (in this case, “nfs”, which is the storage class for utilizing the NFS storage system), and how much you want to request (in this case, just 1 Gig), but you don’t need to know how the storage was implemented.

Note: Different k8s clusters may offer persistent storage that utilize different storage classes. Within TACC, we also have k8s clusters that utilize the `rbd` storage class, for example. Be sure to check with the k8s administrators to see what storage class(es) might be available.

We create this pvc object with the usual `kubectl apply` command:

```
$ kubectl apply -f hello-pvc.yml
persistentvolumeclaim/hello-jstubby-data created
```

Great, with the pvc created, let’s check back on our pods:

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
hello-deployment-9794b4889-mk6qw	1/1	Running	46	46h
hello-deployment-9794b4889-sx6jc	1/1	Running	46	46h
hello-deployment-9794b4889-v2mb9	1/1	Running	46	46h
hello-deployment-9794b4889-vp6mp	1/1	Running	46	46h
hello-pvc-deployment-ff5759b64-sc7dk	1/1	Running	0	45s

Like magic, our “hello-pvc-deployment” now has a running pod without us making any additional API calls to k8s! This is the power of the declarative aspect of k8s. When we created the `hello-pvc-deployment`, we told k8s to always

keep one pod with the properties specified running at all times, if possible, and k8s continues to try and implement our wishes until we instruct it to do otherwise.

Note: You cannot scale a pod with a volume filled by a persistent volume claim.

7.3.8 Exec Commands in a Running Pod

Because the command running within the “hello-pvc-deployment” pod redirected the echo statement to a file, the hello-pvc-deployment-ff5759b64-sc7dk will have no logs. (You can confirm this is the case for yourself using the logs command as an exercise).

In cases like these, it can be helpful to run additional commands in a running pod to explore what is going on. In particular, it is often useful to run shell in the pod container.

In general, one can run a command in a pod using the following:

```
$ kubectl exec <options> <pod_name> -- <command>
```

To run a shell, we will use:

```
$ kubectl exec -it <pod_name> -- /bin/bash
```

The `-it` flags might look familiar from Docker – they allow us to “attach” our standard input and output to the command we run in the container. The command we want to run is `/bin/bash` for a shell.

Let’s exec a shell in our “hello-pvc-deployment-ff5759b64-sc7dk” pod and look around:

```
$ k exec -it hello-pvc-deployment-5b7d9775cb-xspn7 -- /bin/bash
root@hello-pvc-deployment-5b7d9775cb-xspn7:/#
```

Notice how the shell prompt changes after we issue the `exec` command – we are now “inside” the container, and our prompt has changed to “`root@hello-pvc-deployment-5b7d9775cb-xspn`” to indicate we are the root user within the container.

Let’s issue some commands to look around:

```
$ pwd
/
# cool, exec put us at the root of the container's file system

$ ls -l
total 8
drwxr-xr-x  2 root root 4096 Jan 18 21:03 bin
drwxr-xr-x  2 root root   6 Apr 24 2018 boot
drwxr-xr-x  3 root root 4096 Mar  4 01:06 data
drwxr-xr-x  5 root root  360 Mar  4 01:12 dev
drwxr-xr-x  1 root root   66 Mar  4 01:12 etc
drwxr-xr-x  2 root root   6 Apr 24 2018 home
drwxr-xr-x  8 root root  96 May 23 2017 lib
drwxr-xr-x  2 root root  34 Jan 18 21:03 lib64
drwxr-xr-x  2 root root   6 Jan 18 21:02 media
drwxr-xr-x  2 root root   6 Jan 18 21:02 mnt
drwxr-xr-x  2 root root   6 Jan 18 21:02 opt
dr-xr-xr-x 887 root root   0 Mar  4 01:12 proc
```

(continues on next page)

(continued from previous page)

```

drwx----- 2 root root 37 Jan 18 21:03 root
drwxr-xr-x 1 root root 21 Mar 4 01:12 run
drwxr-xr-x 1 root root 21 Jan 21 03:38 sbin
drwxr-xr-x 2 root root 6 Jan 18 21:02 srv
dr-xr-xr-x 13 root root 0 May 5 2020 sys
drwxrwxrwt 2 root root 6 Jan 18 21:03 tmp
drwxr-xr-x 1 root root 18 Jan 18 21:02 usr
drwxr-xr-x 1 root root 17 Jan 18 21:03 var
# as expected, a vanilla linux file system.
# we see the /data directory we mounted from the volume...

$ ls -l data/out.txt
-rw-r--r-- 1 root root 19 Mar 4 01:12 data/out.txt
# and there is out.txt, as expected

$ cat data/out.txt
Hello, Kubernetes!
# and our hello message!

$ exit
# we're ready to leave the pod container

```

Note: To exit a pod from within a shell (i.e., `/bin/bash`) type “exit” at the command prompt.

Note: The `exec` command can only be used to execute commands in *running* pods.

7.3.9 Persistent Volumes Are... Persistent

The point of persistent volumes is that they live beyond the length of one pod. Let’s see this in action. Do the following:

1. Delete the “hello-pvc” pod. What command do you use?
2. After the pod is deleted, list the pods again. What do you notice?
3. What contents do you expect to find in the `/data/out.txt` file? Confirm your suspicions.

Solution.

```

$ kubectl delete pods hello-pvc-deployment-5b7d9775cb-xspn7
pod "hello-pvc-deployment-5b7d9775cb-xspn7" deleted

$ kubectl get pods
NAME                                READY   STATUS              RESTARTS   AGE
hello-deployment-9794b4889-mk6qw    1/1     Running             47         47h
hello-deployment-9794b4889-sx6jc    1/1     Running             47         47h
hello-deployment-9794b4889-v2mb9    1/1     Running             47         47h
hello-deployment-9794b4889-vp6mp    1/1     Running             47         47h
hello-pvc-deployment-5b7d9775cb-7nfhv 0/1     ContainerCreating   0          46s
# wild -- a new hello-pvc-deployment pod is getting created automatically!

```

(continues on next page)

(continued from previous page)

```
# let's exec into the new pod and check it out!
$ k exec -it hello-pvc-deployment-5b7d9775cb-7nfhv -- /bin/bash

$ cat /data/out.txt
Hello, Kubernetes!
Hello, Kubernetes!
```

Warning: Deleting a persistent volume claim deletes all data contained in all volumes filled by the PVC permanently! This cannot be undone and the data cannot be recovered!

7.3.10 Additional Resources

- [Kubernetes Deployments Documentation](#)
- [Persistent Volumes](#)
- [NFS Storage class in k8s](#)
- [Ceph RBD Storage class in k8s](#)

7.4 Services

Services are the k8s resource one uses to expose HTTP APIs, databases and other components that communicate on a network to other k8s pods and, ultimately, to the outside world. To understand services we need to first discuss how k8s networking works.

7.4.1 k8s Networking Overview

Note: We will be covering just the basics of k8s networking, enough for you to become proficient with the main concepts involved in deploying your application. Many details and advanced concepts will be omitted.

k8s creates internal networks and attaches pod containers to them to facilitate communication between pods. For a number of reasons, including security, these networks are not reachable from outside k8s.

Recall that we can learn the private network IP address for a specific pod with the following command:

```
$ kubectl get pods <pod_name> -o wide
```

For example:

```
$ kubectl get pods hello-deployment-9794b4889-mk6qw -o wide
NAME                                READY   STATUS    RESTARTS   AGE   IP
↪NODE   NOMINATED NODE   READINESS GATES
hello-deployment-65c5bbdc69-lk26j   1/1     Running   0           15m   172.16.178.5
↪kube-3.novalocal   <none>   <none>
```

This tells us k8s assigned an IP address of 172.16.178.5 to our hello-deployment pod.

k8s assigns every pod an IP address on this private network. Pods that are on the same network can communicate with other pods using their IP address.

7.4.2 Ports

To communicate with a program running on a network, we use ports. We saw how our flask program used port 5000 to communicate HTTP requests from clients. We can expose ports in our k8s deployments by defining a `ports` stanza in our `template.spec.containers` object. Let's try that now.

Create a file called `hello-flask-deployment.yml` and copy the following contents

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: helloflask
  labels:
    app: helloflask
spec:
  replicas: 1
  selector:
    matchLabels:
      app: helloflask
  template:
    metadata:
      labels:
        app: helloflask
    spec:
      containers:
        - name: helloflask
          imagePullPolicy: Always
          image: jstubbs/hello-flask
          env:
            - name: FLASK_APP
              value: "app.py"
          ports:
            - name: http
              containerPort: 5000
```

Much of this will look familiar. We are creating a deployment that matches the pod description given in the `template.spec` stanza. The pod description uses an image, `jstubbs/hello-flask`. This image runs a very simple flask server that responds with simple text messages to a few endpoints.

The `ports` attribute is a list of k8s port descriptions. Each port in the list includes:

- `name` – the name of the port, in this case, `http`. This could be anything we want really.
- `containerPort` – the port inside the container to expose, in this case `5000`. This needs to match the port that the containerized program (in this case, flask server) is binding to.

Let's create the `helloflask` deployment using `kubectl apply`

```
$ kubectl apply -f hello-flask-deployment.yml
deployment.apps/hello-flask-deployment configured
```

With our deployment created, we should see a new pod.

Exercise. Determine the IP address of the new pod for the hello-flask-deployment.

Solution.

```
$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
hello-deployment-9794b4889-w4jlq    1/1     Running   0           56m
hello-pvc-deployment-6dbbfdc4b4-sxk78 1/1     Running   231         9d
helloflask-86d4c7d8db-2rkg5         1/1     Running   0           5m10s

$ kubectl get pods helloflask-86d4c7d8db-2rkg5 -o wide
NAME                                READY   STATUS    RESTARTS   AGE   IP
↪NODE                               NOMINATED NODE READINESS GATES
hello-deployment-65c5bbdc69-lk26j 1/1     Running   0           18m   172.16.178.5
↪kube-3.novalocal                  <none>   <none>
```

Therefore, the IP address is 172.16.178.5

We found the IP address for our flask container, but if we try to communicate with it from the k8s API node, we will either find that it hangs indefinitely or possibly gives an error:

```
$ curl 172.16.178.5:5000
curl: (7) Failed connect to 172.16.178.5:5000; Network is unreachable
```

This is because the 172.16.*.* private k8s network is not available from the outside, not even from the API node. However, it *is* available from other pods in the namespace.

7.4.3 A Debug Deployment

For exploring and debugging k8s deployments, it can be helpful to have a basic container on the network. We can create a deployment for this purpose.

For example, let's create a deployment using the official python 3.9 image. We can run a sleep command inside the container as the primary command, and then, once the container pod is running, we can use `exec` to launch a shell inside the container.

EXERCISE

1. Create a new “debug” deployment using the following definition:

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: py-debug-deployment
  labels:
    app: py-app
spec:
  replicas: 1
  selector:
    matchLabels:
```

(continues on next page)

(continued from previous page)

```

app: py-app
template:
  metadata:
    labels:
      app: py-app
  spec:
    containers:
      - name: py39
        image: python:3.9
        command: ['sleep', '999999999']

```

(Hint: paste the content into a new file called `deployment-python-debug.yml` and then use the `kubectl apply` command).

2. Exec into the running pod for this deployment. (Hint: find the pod name and then use the `kubectl exec` command, running the shell (`/bin/bash`) command in it).

Once we have a shell running inside our debug deployment pod, we can try to access our flask server. Recall that the IP and port for the flask server were determined above to be `10.244.7.95:5000` (yours will be different).

If we try to access it using `curl` from within the debug container, we get:

```

root@py-debug-deployment-5cc8cdd65f-xzhzq: $ curl 172.16.178.5:5000
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>404 Not Found</title>
<h1>Not Found</h1>
<p>The requested URL was not found on the server. If you entered the URL manually please
↵check your spelling and try again.</p>

```

That's a different error from before, and that's good! This time, the error is from flask, and it indicates that flask doesn't have a route for the root path (`/`).

The `jstubbs/hello-flask` image does not define a route for the root path (`/`) but it does define a route for the path `/hello-service`. If we try that path, we should get a response:

```

root@py-debug-deployment-5cc8cdd65f-xzhzq: $ curl 10.244.7.95:5000/hello-service
Hello world

```

Great! k8s networking from within the private network is working as expected!

7.4.4 Services

We saw above how pods can use the IP address of other pods to communicate. However, that is not a great solution because we know the pods making up a deployment come and go. Each time a pod is destroyed and a new one created it gets a new IP address. Moreover, we can scale the number of replica pods for a deployment up and down to handle more or less load.

How would an application that needs to communicate with a pod know which IP address to use? If there are 3 pods comprising a deployment, which one should it use? This problem is referred to as the *service discovery problem* in distributed systems, and k8s has a solution for it.. the **Service** abstraction.

A k8s service provides an abstract way of exposing an application running as a collection of pods on a single IP address and port. Let's define a service for our `hello-flask` deployment.

Copy and paste the following code into a file called `hello-flask-service.yml`:

```

---
apiVersion: v1
kind: Service
metadata:
  name: hello-service
spec:
  type: ClusterIP
  selector:
    app: helloflask
  ports:
  - name: helloflask
    port: 5000
    targetPort: 5000

```

Let's look at the spec description for this service.

- **type** – There are different types of k8s services. Here we are creating a **ClusterIP** service. This creates an IP address on the private k8s network for the service. We may see other types of k8s services later.
- **selector** – This tells k8s what pod containers to match for the service. Here we are using a label, **app: helloflask**, which means k8s will link all pods with this label to our service. Note that it is important that this label match the label applied to our pods in the deployment, so that k8s links the service up to the correct pods.
- **ports** - This is a list of ports to expose in the service.
- **ports.port** – This is the port to expose on the service's IP. This is the port clients will use when communicating via the service's IP address.
- **ports.targetPort** – This is the port on the pods to target. This needs to match the port specified in the pod description (and the port the containerized program is binding to).

We create this service using the `kubectl apply` command, as usual:

```

$ kubectl apply -f hello-flask-service.yml
service/hello-service configured

```

We can list the services:

```

$ kubectl get services
NAME                TYPE        CLUSTER-IP      EXTERNAL-IP  PORT(S)
--
hello-service       ClusterIP   10.108.58.137   <none>       5000/TCP

```

We see k8s created a new service with IP `10.108.58.137`. We should be able to use this IP address (and port 5000) to communicate with our flask server. Let's try it. Remember that we have to be on the k8s private network, so we need to exec into our debug deployment pod first.

```

$ kubectl exec -it py-debug-deployment-5cc8cdd65f-xzhzq -- /bin/bash

# from inside the container ---
root@py-debug-deployment-5cc8cdd65f-xzhzq:/ $ curl 10.108.58.137:5000/hello-service
Hello world

```

It worked! Now, if we remove our hello-flask pod, k8s will start a new one with a new IP address, but our service will automatically route requests to the new pod. Let's try it.

```
# remove the pod ---
$ kubectl delete pods helloflask-86d4c7d8db-2rkg5
pod "helloflask-86d4c7d8db-2rkg5" deleted

# see that a new one was created ---
$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
hello-deployment-9794b4889-w4jlq    1/1     Running   2           175m
hello-pvc-deployment-6dbbfdc4b4-sxk78 1/1     Running   233         9d
helloflask-86d4c7d8db-vbn4g         1/1     Running   0           62s

# it has a new IP ---
$ kubectl get pods helloflask-86d4c7d8db-vbn4g -o wide
NAME                                READY   STATUS    RESTARTS   AGE   IP             NODE
└─NOMINATED NODE   READINESS GATES
helloflask-86d4c7d8db-vbn4g    1/1     Running   0           112s   10.244.7.96    c05
└─<none>              <none>
# Yep, 10.244.7.96 -- that's different; the first pod had IP 10.244.7.95

# but back in the debug deployment pod, check that we can still use the service IP --
root@py-debug-deployment-5cc8cdd65f-xzhzq:/ $ curl 10.108.58.137:5000/hello-service
Hello world
```

Note that k8s is doing something non-trivial here. Each pod could be running on one of any number of worker computers in the TACC k8s cluster. When the first pod was deleted and k8s created the second one, it is quite possible it started it on a different machine. So k8s had to take care of rerouting requests from the service to the new machine.

k8s can be configured to do this “networking magic” in different ways. While the details are beyond the scope of this course, keep in mind that the virtual networking that k8s uses does come at a small cost. For most applications, including long-running web APIs and databases, this cost is negligible and isn’t a concern. But for high-performance applications, and in particular, applications whose performance is bounded by the performance of the underlying network, the overhead can be significant.

7.4.5 HomeWork 6 – Deploying Our Flask API to k8s

In this section we will use class time to deploy our Flask API to k8s. This will be a guided, hands-on lab, and it will also be submitted for a grade as HomeWork 6. Feel free to ask questions as you work through the lab. Any part of the assignment that you do not complete during class will need to be completed outside of class before you submit.

Our goal is to create a “test” environment for our Flask API application. We will be using names and labels accordingly. Later in the semester, you will create a “production” environment for your Flask API application as well. You can use this lab as a guide to do that.

In each step you will create a k8s object described in a separate yaml file. Name the files `<username>-<env>-<app>-<kind>.yaml`. Use “test” for `<env>` since we are creating the test environment. For example, my Redis deployment would be called `jstubbs-test-redis-deployment.yaml` while my redis service would be called `jstubbs-test-redis-service.yaml`.

Step 1. We will start by focusing on our Redis container. Our Flask API depends on Redis so it makes sense to start there. Since Redis writes our application data to disk, we will need a way to save the data independent of the Redis pods. Create a persistent volume claim for your Redis data. Use the following information when creating your PVC:

- The name of your PVC should include your TACC username and the word “test”, to indicate it is in the test environment.

- We'll make use of `labels` to add additional metadata to our k8s objects that will help us search and filter them. Let's add a `username` label and an `env` label. The value for `username` should be your tacc username and the value for `env` should be `test`, to indicate that this is the test environment.
- The `accessModes` should include a single entry, `ReadWriteOnce`.
- The `storageClassName` should be `nfs`.
- Be sure to request 1 GB (1Gi) of storage.

Step 2. Create a deployment for the Redis database. Be sure to include the following:

- The name of your redis deployment should include your TACC username and the word "test", to indicate it is in the test environment.
- Use the same `username` and `env` labels for both the deployment and the pod template.
- Be sure to set `replicas: 1` as Redis is a stateful application.
- For the image, use `redis:6`; you do not need to set a command.
- Add the `username` and `env` labels to the pod as well. Also add an `app` label with value `<username>-test-redis`. This will be important in the next step.
- Be sure to create a `volumeMount` and associate it with a `volume` that is filled by the PVC you created in Step 1. For the mount path, use `/data`, as this is where Redis writes its data.

Step 3. Create a service for your Redis database. This will give you a persistent IP address to use to talk to Redis, regardless of the IPs that may be assigned to individual Redis pods. Be sure to include the following:

- The name of your redis service should include your TACC username and the word "test", to indicate it is in the test environment.
- Use the same `username` and `env` labels for both the deployment and the pod template.
- The `type` of service should be `ClusterIP`.
- Define a `selector` that will select your Redis pods and only your redis pods. What label should you use? Hint: the `env` and `username` labels won't be unique enough.
- Make sure `port` and `targetPort` match the Redis port.

Once you are done with Steps 1 through 3, check your work:

- Look up the service IP address for your test redis service.
- Exec into a Python debug container.
- Install the redis python library.
- Launch the python shell and import redis
- Create a Python redis client object using the IP and port of the service, something like: `rd = redis.StrictRedis(host='10.101.101.139', port=6379, db=0)`
- Create a key and make sure you can get the key.
- In another shell on `isp02`, delete the redis pod. Check that k8s creates a new redis pod.
- Back in your python shell, check that you can still get the key using the same IP. This will show that your service is working and that your Redis database is persisting data to the PVC (i.e., the data are surviving pod restarts).

Step 4. Create a deployment for your flask API. If it helps, you can use your Redis deployment as a starting point. Be sure to:

- The name of your flask service should include your TACC username and the word "test", to indicate it is in the test environment.

- Use the same `username` and `env` labels for both the deployment and the pod template.
- start 2 replicas of your flask API pod.
- Be sure to expose port 5000.

Step 5. Create a service for your flask API. This will give you a persistent IP address to use to talk to your flask API, regardless of the IPs that may be assigned to individual flask API pods. Be sure to include the following:

- The name of your redis service should include your TACC username and the word “test”, to indicate it is in the test environment.
- Use the same `username` and `env` labels for both the deployment and the pod template.
- The `type` of service should be `ClusterIP`.
- Define a `selector` that will select your flask API pods and only your flask API pods.
- Make sure `port` and `targetPort` match the flask port.

7.5 k8s Cheat Sheet

This all-in-one k8s cheat sheet can be used for quick reference.

7.5.1 k8s Resource Types

Here are the primary k8s resource types we have covered in this class:

- **Pods** – Pods are the simplest unit of compute in k8s and represent a generalization of the Docker container. Pods can contain more than one container, and every container within a pod is scheduled together, on the same machine, with a single IP address and shared file system. Pod definitions include a `name`, a (Docker) `image`, a `command` to run, as well as an `ImagePullPolicy`, `volumeMounts` and a set of `ports` to expose. Pods can be thought of as disposable and temporary.
- **Deployments** – Deployments are the k8s resource type to use for deploying *long-running* application components, such as APIs, databases, and long-running worker programs. Deployments are made up of one or more matching pods, and the matching is done using `labels` and `labelSelectors`.
- **PersistentVolumeClaims (PVCs)** – PVCs create a named storage request against a storage class available on the k8s cluster. PVCs are used to fill volumes with permanent storage so that data can be saved across different pod executions for the same stateful application (e.g., a database).
- **Services** – Services are used to expose an entire application component to other pods. Services get their own IP address which persists beyond the life of the individual pods making up the application.

7.5.2 kubectl Commands

Here we collect some of the most commonly used `kubectl` commands for quick reference.

Command	Description	Example
kubectl get <resource_type>	List all objects of a given resource type.	kubectl get pods
kubectl get <type> <name>	Get one object of a given type by name.	kubectl get pods hello-pod
kubectl get <type> <name> -o wide	Show additional details of an object	kubectl get pods hello-pod -o wide
kubectl describe <type> <name>	Get full details of an object by name.	kubectl describe pods hello-pod
kubectl logs <name>	Get the logs of a running pod by name.	kubectl logs hello-pod
kubectl logs -f <name>	Tail the logs of a running pod by name.	kubectl logs -f hello-pod
kubectl logs --since <time> <name>	Get the logs of a running pod since a given time.	kubectl logs --since 1m -f hello-pod
kubectl exec -it <name> -- <cmd>	Run a command, <cmd>, in a running pod.	kubectl exec -it hello-pod -- /bin/bash
kubectl apply -f <file>	Create or update an object description using a file.	kubectl apply -f hello-pod.yml

UNIT 8: ASYNCHRONOUS PROGRAMMING

In this unit, we will discuss concurrency and asynchronous programming with the goal of covering the material you will need to add a “long running” jobs functionality to your Flask API project.

8.1 Concurrency and Queues

We begin with an introduction to concurrency and a basic treatment of the queue data structure. By the end of this the module, the student should be able to:

- Describe concurrency and some basic examples of concurrent and nonconcurrent algorithms.
- Explain at a high level how concurrency will be used to implement a long-running task in our flask-based API system.
- Utilize Python in-memory queues as well as the `hotqueue` library to work with queues in Redis.

8.1.1 Motivation

Our Flask API is useful because it can return information about objects in our database, and in general, looking up or storing objects in the database is a very “fast” operation, on the order of a few 10s of milliseconds. However, many interesting and useful operations are not nearly as quick to perform. They are many examples from both research computing and industrial computing where the computations take much longer; for example, on the order of minutes, hours, days or even longer.

Examples include:

- Executing a large mathematical model simulating a major weather event such as a hurricane, or an astronomical process such as galaxy formation
- Aligning a set of genomic sequence fragments to a reference genome
- Running the payroll program at the end of the month to send checks to all employees of a large enterprise.
- Sending a “welcome back” email to every student enrolled at the university at the start of the semester.

We want to be able to add functionality like this to our API system. We’d like to provide a new API endpoint where a user could describe some kind of long-running computation to be performed and have our system perform it for them. But there are a few issues:

- The HTTP protocol was not built for long-running tasks, and most programs utilizing HTTP expect responses “soon”, on the order of a few seconds. Many programs have hard timeouts around 30 or 60 seconds.
- The networks on which HTTP connections are built can be interrupted (even just briefly) over long periods of time. If a connection is severed, even for a few milliseconds, what happens to the long-running computation?

- Long-running tasks like the ones above can be computationally intensive and require a lot of computing resources. If our system becomes popular (even with a single, enthusiastic user), we may not be able to keep up with demand. We need to be able to throttle the number of computations we do.

To address these challenges, we will implement a “Jobs API” as an *asynchronous* endpoint. Over the next few lectures, we will spell out precisely what this means, but for now, we’ll give a quick high-level overview as motivation. Don’t worry about understanding all the details here.

8.1.2 Jobs API – An Introduction

The basic idea is that we will have a new endpoint in our API at a path `/jobs` (or something similar). A user wanting to have our system perform a long-running task will create a new job. We will use RESTful semantics, so the user will create a new job by making an HTTP POST request to `/jobs`, describing the job in the POST message body (in JSON).

However, instead of performing the actual computation, the Jobs API will simply record that the user has requested such a computation. It will store that in Redis and immediately respond to the user. So, the response will not include the result of the job itself but instead it will indicate that the request has been received and it will be worked on in due time. Also, and critically, it will provide an `id` for the job that the user can use to check the status later and, eventually, get the actual result.

So, in summary:

1. User makes an HTTP POST to `/jobs` to create a job.
2. Jobs API validates that the job is a valid job, creates an `id` for it, and stores the job description with the `id` in Redis.
3. Jobs API responds to the user immediately with the `id` of the job it generated.
4. In the background, *some other python program* we write (referred to as a “worker”) will, at some point in the future, actually start the job and monitor it to completion.

This illustrates the *asynchronous* and *concurrent* nature of our Jobs API, terms we will define precisely in the sequel. Intuitively, you can probably already imagine what we mean here – multiple jobs can be worked on at the same time by different instances of our program (i.e., different workers), and the computation happens asynchronously from the original user’s request.

8.1.3 Concurrency and Queues

A computer system is said to be *concurrent* if multiple agents or components of the system can be in progress at the same time without impacting the correctness of the system.

While components of the system are in progress at the same time, the individual operations themselves may happen sequentially. In general, a system being concurrent means that the different components can be executed at the same time or in different orders without impacting the overall correctness of the system.

There are many techniques for making programs concurrent; we will primarily focus on a technique that leverages the *queue* data structure. But first, an example.

A First Example

Suppose we want to build a system for maintaining the balance of a bank account where multiple agents are acting on the account (withdrawing and/or depositing funds) at the same time. We will consider two different approaches.

Approach 1. Whenever an agent receives an order to make a deposit or withdraw, the agent does the following steps:

1. Makes a query to determine the current balance.
2. Computes the new balance based on the deposit or withdraw amount.
3. Makes a query to update the balance to the computed amount.

This approach is not concurrent because the individual operations of different agents cannot be reordered.

For example, suppose we have:

- Two agents, agent A and agent B, and a starting balance of \$50.
- Agent A gets an order to deposit \$25 at the same time that agent B gets an order to withdraw \$10.

In this case, the final balance should be \$65 ($=\$50 + \$25 - \10).

The system will arrive at this answer as long as steps 1, 2 and 3 for one agent are done before any steps for the other agent are started; for ex, A1, A2, A3, B1, B2, B3.

However, if the steps of the two agents are mixed then the system will not arrive at the correct answer.

For example, suppose the steps of the two agents were performed in this order: A1, A2, B1, B2, A3, B3. What would the final result be? The listing below shows what each agents sees at each step.

- A1. Agent A determines the current balance to be \$50.
- A2. Agent A computes a new balance of $\$50 + \$25 = \$75$.
- B1. Agent B determines the current balance to be \$50.
- B2. Agent B computes a new balance of $\$50 - \$10 = \$40$.
- A3. Agent A updates the balance to be \$75.
- B3. Agent B updates the balance to be \$40.

In this case, the system will compute the final balance to be \$40! Hopefully this is not your account! :)

We will explore an alternative approach that is concurrent, but to do that we first need to introduce the concept of a queue.

Queues

A queue is data structure that maintains an ordered collection of items. The queue typically supports just two operations:

- Enqueue (aka “put”) - add a new item to the queue.
- Dequeue (aka “get”) - remove an item from the queue.

Items are removed from a queue in First-In-First-Out (FIFO) fashion: that is, the item removed from the first dequeue operation will be the first item added to the queue, the item removed from the second dequeue operation will be the second item added to the queue, and so on.

Sometimes queues are referred to as “FIFO Queues” for emphasis.

Basic Queue Example

Consider the set of (abstract) operations on a Queue object.

```
1. Enqueue 5
2. Enqueue 7
3. Enqueue A
4. Dequeue
5. Enqueue 1
6. Enqueue 4
7. Dequeue
8. Dequeue
```

The order of items returned is:

```
5, 7, A
```

And the contents of the Queue after Step 8 is

```
1, 4
```

In-memory Python Queues

The Python standard library provides an in-memory Queue data structure via its `queue` module. To get started, import the `queue` module and instantiate a `queue.Queue` object:

```
>>> import queue
>>> q = queue.Queue()
```

The Python Queue object has the following features:

- The `q` object supports `.put()` and `.get()` to put a new item on the queue, and get an item off the queue, respectively
- `q.put()` can take an arbitrary Python object and `q.get()` returns a Python object from the queue.

Let's perform the operations above using the `q` object.

Exercise. Use a series of `q.put()` and `q.get()` calls to perform Steps 1-8 above. Verify the the order of items returned.

Exercise. Verify that arbitrary Python objects can be put onto and retrieved from the queue by inserting a list and a dictionary.

Queues are a fundamental ingredient in concurrent programming, a topic we will turn to next.

A Concurrent Approach to Our Example

Approach 2. Whenever an agent receives an order to make a withdraw or deposit, the agent simply writes the order to a queue; a positive number indicates a deposit while a negative number indicates a withdraw. The account system keeps a running “balancer” agent whose only job is to read items off the queue and update the balance.

This approach is concurrent because the order of the agents’ steps can be mixed without impacting the overall result. This fact essentially comes down to the commutativity of addition and subtraction operations: i.e., $50 + 25 - 10 = 50 - 10 + 25$.

Note that the queue of orders could be generalized to a “queue of tasks” (transfer some amount from account A to account B, close account C, etc.).

Queues in Redis

The Python in-memory queues are very useful for a single Python program, but we ultimately want to share queues across multiple Python programs/containers.

The Redis DB we have been using can also be used to provide a queue data structure for clients running in different containers. The basic idea is:

- Use a Redis list data structure to hold the items in the queue.
- Use the Redis list operations `rpush`, `lpop`, `llen`, etc. to create a queue data structure.

For example:

- `rpush` will add an element to the end of the list.
- `lpop` will return an element from the front of the list, and return nothing if the list is empty.
- `llen` will return the number of elements in the list.

Fortunately, we don’t have to implement the queue ourselves, but know that if we needed to we could without too much effort.

Using the hotqueue library

We will leverage a small, open source Python library called `hotqueue` which has already implemented the a Queue data structure in Redis using the approach outlined above. Besides not having to write it ourselves, the use of `hotqueue` will afford us a few additional features which we will look at later.

Here are the basics of the `hotqueue` library:

- Hotqueue is not part of the Python standard library; you can install it with `pip install hotqueue`
- Creating a new queue data structure or connecting to an existing queue data structure is accomplished by creating a `HotQueue` object.
- Constructing a `HotQueue` object takes very similar parameters to that of the `StrictRedis` but also takes a `name` attribute. The `HotQueue` object ultimately provides a connection to the Redis server.
- Once constructed, a `HotQueue` object has `.put()` and `.get()` methods that act just like the corresponding methods of an in-memory Python queue.

A Hotqueue Example

We will work this example out on the k8s cluster. You will need a Redis pod running on the cluster and you will also need the python debug pod you created last lecture.

If you prefer, you can create a new deployment that uses the `jstubbs/redis-client` image with the required libraries already installed using the following code –

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-client-debug-deployment
  labels:
    app: redis-client-debug
spec:
  replicas: 1
  selector:
    matchLabels:
      app: redis-client-debug
  template:
    metadata:
      labels:
        app: redis-client-debug
    spec:
      containers:
        - name: py39
          image: jstubbs/redis-client
          command: ['sleep', '999999999']
```

With your debug pod running, first, exec into it and install `redis` and `hotqueue`. You can optionally also install `ipython` which is a nicer Python REPL (Read, Evaluate, Print Loop).

Note: The `jstubbs/redis-client` image has these libraries already installed.

```
$ kubectl get pods -o wide
```

	NAME	READY	STATUS	RESTARTS	AGE	IP
→	hello	1/1	Running	199	8d	10.244.5.
→	214 c04 <none> <none>					
	hello-deployment-55f4459bf-npdrn	1/1	Running	79	3d7h	10.244.5.5
→	c04 <none> <none>					
	hello-pvc-deployment-6dbbfdc4b4-whjwb	1/1	Running	31	31h	10.244.3.
→	143 c01 <none> <none>					
	helloflask-848c4fb54f-9j4fd	1/1	Running	0	30h	10.244.3.
→	188 c01 <none> <none>					
	helloflask-848c4fb54f-gpqhb	1/1	Running	0	30h	10.244.5.
→	75 c04 <none> <none>					
	jstubbs-test-redis-64cbc6b8cf-f6qr1	1/1	Running	0	3m5s	10.244.3.
→	237 c01 <none> <none>					
	py-debug-deployment-5cc8cdd65f-tr9gq	1/1	Running	0	31h	10.244.3.
→	177 c01 <none> <none>					

(continues on next page)

(continued from previous page)

```
$ kubectl exec -it py-debug-deployment-5cc8cdd65f-tr9gq -- /bin/bash
$ pip install redis hotqueue ipython
```

Start the python (or ipython) shell and create the `hotQueue.Queue` object. You can use the Redis IP directly, or use the Redis service IP if you create one.

```
>>> from hotqueue import HotQueue
>>> q = HotQueue("queue", host="<Redis_IP>", port=6379, db=1)
```

Note how similar the `HotQueue()` instantiation is to the `StrictRedis` instantiation. In the example above we named the queue `queue` (not too creative), but it could have been anything.

Note: In the definition above, we have set `db=1` to ensure we don't interfere with the main data of your Flask app.

Now we can add elements to the queue using the `.put()`; just like with in-memory Python queues, we can put any Python object into the queue:

```
>>> q.put(1)
>>> q.put('abc')
>>> q.put(['1', 2, {'key': 'value'}, '4'])
```

We can check the number of items in queue at any time using the `len` built in:

```
>>> len(q)
3
```

And we can remove an item with the `.get()` method; remember - the queue follows a FIFO principle:

```
>>> q.get()
1
>>> len(q)
2
>>> q.get()
'abc'
>>> len(q)
1
```

Under the hood, the `hotqueue.Queue` is just a Redis object, which we can verify using a redis client:

```
>>> import redis
>>> rd = redis.StrictRedis(host="<Redis_IP>", port=6379, db=1)
>>> rd.keys()
[b'hotqueue:queue']
```

Note that the queue is just a single key in the Redis server (`db=1`).

And just like with other Redis data structures, we can connect to our queue from additional Python clients and see the same data.

Exercise. In a second SSH shell, scale your Python debug deployment to 2 replicas, install redis, hotqueue, and ipython in the new replica, start iPython and connect to the same queue. Prove that you can use `get` and `put` to “communicate” between your two Python programs.

8.2 Messaging Systems

The Queue is a powerful data structure which forms the foundation of many concurrent design patterns. Often, these design patterns center around passing messages between agents within the concurrent system. We will explore one of the simplest and most useful of these message-based patterns - the so-called “Task Queue”. Later, we may also look at the somewhat related “Publish-Subscribe” pattern (also sometimes referred to as “PubSub”).

By the end of this module, the student should be able to:

- Describe the components of a task queue system, and explain how it will be utilized within our Flask-based API system architecture.
- Create task queues in Redis using the `hotqueue` library, and work with the `put()` and `consume()` methods to queue and receive messages across two Python programs.
- Use the `q.worker` decorator in `hotqueue` to create a simple Python consumer program.
- Explain the general approach to organizing Python code into different modules and describe how to do this for the flask-based API system we are building.
- Implement good code organization practices including denoting objects as public or private.

8.2.1 Task Queue (or Work Queue)

In a task queue system,

- Agents called “producers” write messages to a queue that describe work to be done.
- A separate set of agents called “consumers” receive the messages and do the work. While work is being done, no new messages are received by the consumer.
- Each message is delivered exactly once to a single consumer to ensure no work is “duplicated”.
- Multiple consumers can be processing “work” messages at once, and similarly, 0 consumers can be processing messages at a given time (in which case, messages will simply queue up).

The Task Queue pattern is a good fit for our jobs service.

- Our Flask API will play the role of producer.
- One or more “worker” programs will play the role of consumer.
- Workers will receive messages about new jobs to execute and performing the analysis steps.

8.2.2 Task Queues in Redis

The `HotQueue` class provides two methods for creating a task queue consumer; the first is the `.consume()` method and the second is the `q.worker` decorator.

The Consume Method

With a `q` object defined like `q = HotQueue("some_queue", host="<Redis_IP>", port=6379, db=1)`, the consume method works as follows:

- The `q.consume()` method returns an iterator which can be looped over using a `for` loop (much like a list).
- Each object returned by the iterator is a message received from the task queue.
- The `q.consume()` method blocks (i.e., waits indefinitely) when there are no additional messages in the queue named `some_queue`.

The basic syntax of the consume method is this:

```
for item in q.consume():
    # do something with item
```

In this case, the `item` object is the message that was retrieved from the task queue.

Exercises. Complete the following, either in Kubernetes or directly on isp02. (see k8s files below.)

1. Start/scale two python debug containers with redis and hotqueue installed (you can use the `jstubbs/redis-client` image if you prefer). In two separate shells, `exec` into each debug container and start `ipython`.
2. In each terminal, create a `HotQueue` object pointing to the same Redis queue.
3. In the first terminal, add three or four Python strings to the queue; check the length of the queue.
4. In the second terminal, use a `for` loop and the `.consume()` method to print objects in the queue to the screen.
5. Observe that the strings are printed out in the second terminal.
6. Back in the first terminal, check the length of the queue; add some more objects to the queue.
7. Confirm the newly added objects are “instantaneously” printed to the screen back in the second terminal.

If you want, you can use the following k8s files for the exercise above (but if you already have a redis deployment, you don't need to create a new one.)

Content for the `redis-client-debug-deployment.yml` file:

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-client-debug-deployment
  labels:
    app: redis-client-debug
spec:
  replicas: 2
  selector:
    matchLabels:
      app: redis-client-debug
  template:
    metadata:
      labels:
        app: redis-client-debug
    spec:
      containers:
        - name: py39
```

(continues on next page)

(continued from previous page)

```
image: jstubbs/redis-client
command: ['sleep', '999999999']
```

Content for the `redis-ex-deployment.yml` file:

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: jstubbs-redis-ex-deployment
  labels:
    app: jstubbs-redis-ex
spec:
  replicas: 1
  selector:
    matchLabels:
      app: jstubbs-redis-ex
  template:
    metadata:
      labels:
        app: jstubbs-redis-ex
    spec:
      containers:
        - name: jstubbs-test-redis
          image: redis:6
```

The `q.worker` Decorator

Given a Hotqueue queue object, `q`, the `q.worker` decorator is a convenience utility to turn a function into a consumer without having to write the for loop. The basic syntax is:

```
@q.worker
def do_work(item):
    # do something with item
```

In the example above, `item` will be populated with the item dequeued.

Then, to start consuming messages, simply call the function:

```
>>> do_work()
# ... blocks until new messages arrive
```

Note: The `@q.worker` decorator replaces the for loop. Once you call a function decorated with `@q.worker`, the code never returns unless there is an unhandled exception.

Exercise. Write a function, `echo(item)`, to print an item to the screen, and use the `q.worker` decorator to turn it into a consumer. Call your `echo` function in one terminal and in a separate terminal, send messages to the redis queue. Verify that the message items are printed to the screen in the first terminal.

In practice, we will use the `@q.worker` in a Python source file like so –

```
# A simple example of Python source file, worker.py
q = HotQueue("some_queue", host="<Redis_IP>", port=6379, db=1)

@q.worker
def do_work(item):
    # do something with item...

do_work()
```

Assuming the file above was saved as `worker.py`, calling `python worker.py` from the shell would result in a non-terminating program that “processed” the items in the “some_queue” queue using the `do_work(item)` function. The only thing that would cause our worker to stop is an unhandled exception.

8.2.3 Concurrency in the Jobs API

Recall that our big-picture goal is to add a Jobs endpoint to our Flask system that can process long-running tasks. We will implement our Jobs API with concurrency in mind. The goals will be:

- Enable analysis jobs that take longer to run than the request/response cycle (typically, a few seconds or less).
- Deploy multiple “worker” processes to enable more throughput of jobs.

The overall architecture will thus be:

- a) Save the request in a database and respond to the user that the analysis will eventually be run.
- b) Give the user a unique identifier with which they can check the status of their job and fetch the results when they are ready,
- c) Queue the job to run so that a worker can pick it up and run it.
- d) Build the worker to actually work the job.

Parts a), b) and c) are the tasks of the Flask API, while part d) will be a worker, running as a separate pod/container, that is waiting for new items in the Redis queue.

8.2.4 Code Organization

As software systems get larger, it is very important to keep code organized so that finding the functions, classes, etc. responsible for different behaviors is as easy as possible. To some extent, this is technology-specific, as different languages, frameworks, etc., have different rules and conventions about code organization. We’ll focus on Python, since that is what we are using.

The basic unit of code organization in Python is called a “module”. This is just a Python source file (ends in a `.py` extension) with variables, functions, classes, etc., defined in it. We’ve already used a number of modules, including modules that are part of the Python standard library (e.g. `json`) and modules that are part of third-party libraries (e.g., `redis`).

The following should be kept in mind when designing the modules of a larger system:

- Modules should be focused, with specific tasks or functionality in mind, and their names (preferably, short) should match their focus.
- Modules are also the most typical entry-point for the Python interpreter itself, (e.g., `python some_module.py`).
- Accessing code from external modules is accomplished through the `import` statement.
- Circular imports will cause errors - if module A imports an object from module B, module B cannot import from module A.

Examples. The Python standard library is a good source of examples of module design. You can browse the standard library for Python 3.9 [here](#).

- We see the Python standard library has modules focused on a variety of computing tasks; for example, for working with different data types, such as the `datetime` module and the `array` module. The descriptions are succinct:
 - *The `datetime` module supplies classes for manipulating dates and times.*
 - *This module defines an object type which can compactly represent an array of basic values: characters, integers, floating point numbers*
- For working with various file formats: e.g., `csv`, `configparser`
- For working with concurrency: `threading`, `multiprocessing`, etc.

With this in mind, a first approach might be to break up our system into two modules:

- `api.py` - this module contains the flask web server.
- `worker.py` - this module contains the code to execute jobs.

However, both the API server and the workers will need to interact with the database and the queue:

- The API will create new jobs in the database, put new jobs onto the queue, and retrieve the status of jobs (and probably the output products of the job).
- The worker will pull jobs off the queue, retrieve jobs from the database, and update them.

This suggests a different structure:

- `api.py` - this module contains the flask web server.
- `jobs.py` - this module contains core functionality for working with jobs in Redis (and on the queue).
- `worker.py` - this module contains the code to execute jobs.

Common code for working with `redis/hotqueue` can go in the `jobs.py` module and be imported in both `api.py` and `worker.py`.

Note: High-quality modular design is a crucial aspect of building good software. It requires significant thought and experience to do correctly, and when done poorly it can have dire consequences. In the best case, poor module design can make the software difficult to maintain/upgrade; in the worst case, it can prevent it from running correctly at all.

We can sketch out our module design by making a list of the functionality that will be available in each module. This is only an initial pass at listing the functionality needed – we will refine it over time – but making an initial list is important for thinking through the problem.

`api.py`: This file will contain all the functionality related to the flask web server, and will include functions related to each of the API endpoints in our application.

- `POST /data` – Load the data into the application. Will write to Redis.
- `GET /data?search=...` – List all of the data in the system, optionally filtering with a search query parameter. Will read from Redis.
- `GET /data/<id>` – Get a specific object from the dataset using its id. Will read from Redis.
- `POST /jobs` – Create a new job. This function will save the job description to Redis and add a new task on the queue for the job. Will write to Redis and the queue.
- `GET /jobs` – List all the jobs. Will read from Redis.
- `GET /jobs/<id>` – Get the status of a specific job by id. Will read from Redis.
- `GET /jobs/<id>/results` – Return the outputs (results) of a completed job. Will read from Redis.

`worker.py`: This file will contain all of the functionality needed to get jobs from the task queue and execute the jobs.

- Get a new job – Hotqueue consumer to get an item off the queue. Will get from the queue and write to Redis to update the status of the job.
- Perform analysis –
- Finalize job – Saves the results of the analysis and updates the job status to complete. Will write to Redis.

`jobs.py`: This file will contain all functionality needed for working with jobs in the Redis database and the Hotqueue queue.

- Save a new job – Will need to write to Redis.
- Retrieve an existing job - Will need to read from Redis.
- Update an existing jobs – Will need to read and write to Redis.

8.2.5 Private vs Public Objects

As software projects grow, the notion of public and private access points (functions, variables, etc.) becomes an increasingly important part of code organization.

- Private objects should only be used within the module they are defined. If a developer needs to change the implementation of a private object, she only needs to make sure the changes work within the existing module.
- Public objects can be used by external modules. Changes to public objects need more careful analysis to understand the impact across the system.

Like the layout of code itself, this topic is technology-specific. In this class, we will take a simplified approach based on our use of Python. Remember, this is a simplification to illustrate the basic concepts - in practice, more advanced/robust approaches are used.

- We will name private objects starting with a single underscore (`_`) character.
- If an object does not start with an underscore, it should be considered public.

We can see public and private objects in use within the standard library as well. If we open up the source code for the `datetime` module, which can be found [on GitHub](#) we see a mix of public and private objects and methods.

- Private objects are listed first.
- Public objects start on [line 473](#) with the `timedelta` class.

Exercise. Create three files, `api.py`, `worker.py` and `jobs.py` in your local repository, and update them by working through the following example.

Here are some function and variable definitions, some of which have incomplete implementations and/or have invalid syntax.

To begin, place them in the appropriate files. Also, determine if they should be public or private.

```
def generate_jid():
    """
    Generate a pseudo-random identifier for a job.
    """
    return str(uuid.uuid4())

app = Flask(__name__)

def generate_job_key(jid):
```

(continues on next page)

(continued from previous page)

```

"""
    Generate the redis key from the job id to be used when storing, retrieving or
    ↪ updating
    a job in the database.
"""
    return 'job.{}'.format(jid)

q = HotQueue("queue", host='172.17.0.1', port=6379, db=1)

def instantiate_job(jid, status, start, end):
    """
        Create the job object description as a python dictionary. Requires the job id,
        ↪ status,
        start and end parameters.
    """
    if type(jid) == str:
        return {'id': jid,
                'status': status,
                'start': start,
                'end': end
               }
    return {'id': jid.decode('utf-8'),
            'status': status.decode('utf-8'),
            'start': start.decode('utf-8'),
            'end': end.decode('utf-8')}

@app.route('/jobs', methods=['POST'])
def jobs_api():
    """
        API route for creating a new job to do some analysis. This route accepts a JSON
        ↪ payload
        describing the job to be created.
    """
    try:
        job = request.get_json(force=True)
    except Exception as e:
        return True, json.dumps({'status': "Error", 'message': 'Invalid JSON: {}'.format(e)})
    ↪ return json.dumps(jobs.add_job(job['start'], job['end']))

def save_job(job_key, job_dict):
    """Save a job object in the Redis database."""
    rd.hset(.....)

def queue_job(jid):
    """Add a job to the redis queue."""
    ....

if __name__ == '__main__':
    """
        Main entrypoint of the API server
    """

```

(continues on next page)

(continued from previous page)

```

"""
app.run(debug=True, host='0.0.0.0')

def add_job(start, end, status="submitted"):
    """Add a job to the redis queue."""
    jid = generate_jid()
    job_dict = instantiate_job(jid, status, start, end)
    save_job(.....)
    queue_job(.....)
    return job_dict

@<...> # fill in
def execute_job(jid):
    """
    Retrieve a job id from the task queue and execute the job.
    Monitors the job to completion and updates the database accordingly.
    """
    # fill in ...
    # the basic steps are:
    # 1) get job id from message and update job status to indicate that the job has
    ↪ started
    # 2) start the analysis job and monitor it to completion.
    # 3) update the job status to indicate that the job has finished.

rd = redis.StrictRedis(host='172.17.0.1', port=6379, db=0)

def update_job_status(jid, status):
    """Update the status of job with job id `jid` to status `status`."""
    job = get_job_by_id(jid)
    if job:
        job['status'] = status
        save_job(generate_job_key(jid), job)
    else:
        raise Exception()

```

Solution. We start by recognizing that `app = Flask(__name__)` is the instantiation of a Flask app, the `@app.route` is a flask decorator for defining an endpoint in the API, and the `app.run` line is used to launch the flask server, so we add those both in the `api.py` file:

```

# api.py

app = Flask(__name__)

@app.route('/jobs', methods=['POST'])
def jobs_api():
    """
    API route for creating a new job to do some analysis. This route accepts a JSON
    ↪ payload
    describing the job to be created.
    """
    try:
        job = request.get_json(force=True)

```

(continues on next page)

(continued from previous page)

```

    except Exception as e:
        return True, json.dumps({'status': "Error", 'message': 'Invalid JSON: {}'.format(e)})
    return json.dumps(jobs.add_job(job['start'], job['end']))

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0')

```

We also recognize that several functions appear to be jobs-related:

- generate_jid
- generate_job_key
- instantiate_job
- save_job
- queue_job
- add_job
- execute_job
- update_job_status

Note that the `jobs_api()` function, which we just put in `api.py`, actually references `jobs.add_job`, so we can put `add_job` in the `jobs.py` file as a public function, and anything that it calls can be added to `jobs.py` as a (potentially private) function. Note that `add_job` calls the following functions:

- generate_jid
- instantiate_job
- save_job
- queue_job

so we can put all of these in `jobs.py`:

```

# jobs.py
def generate_jid():
    """
    Generate a pseudo-random identifier for a job.
    """
    return str(uuid.uuid4())

def instantiate_job(jid, status, start, end):
    """
    Create the job object description as a python dictionary. Requires the job id,
    status,
    start and end parameters.
    """
    if type(jid) == str:
        return {'id': jid,
                'status': status,
                'start': start,
                'end': end
               }

```

(continues on next page)

(continued from previous page)

```

    return {'id': jid.decode('utf-8'),
            'status': status.decode('utf-8'),
            'start': start.decode('utf-8'),
            'end': end.decode('utf-8')}

def save_job(job_key, job_dict):
    """Save a job object in the Redis database."""
    rd.hset(.....)

def queue_job(jid):
    """Add a job to the redis queue."""
    ....

def add_job(start, end, status="submitted"):
    """Add a job to the redis queue."""
    jid = _generate_jid()
    job_dict = instantiate_job(jid, status, start, end)
    save_job(.....)
    queue_job(.....)
    return job_dict

```

That leaves the following:

- `q = HotQueue(..)`
- `rd = StrictRedis(..)`
- `update_job_status()`
- `generate_job_key`
- `execute_job()`

Consider that:

- We know `worker.py` is responsible for actually executing the job, so `execute_job` should go there.
- The `update_job_status()` is a jobs-related task, so it goes in the `jobs.py` file – it also makes a call to `instantiate_job` which is already in `jobs.py`.
- The `jobs.py` file definitely needs access to the `rd` object so that goes there.
- Lastly, the `q` will be needed by both `jobs.py` and `worker.py`, but `worker.py` is already importing from `jobs`, so we better put it in `jobs.py` as well.

Therefore, the final placement of all the functions looks like the following:

```

# api.py

app = Flask(__name__)

@app.route('/jobs', methods=['POST'])
def jobs_api():
    """
    API route for creating a new job to do some analysis. This route accepts a JSON_
    ↪payload
    describing the job to be created.
    """

```

(continues on next page)

(continued from previous page)

```

        """
    try:
        job = request.get_json(force=True)
    except Exception as e:
        return True, json.dumps({'status': "Error", 'message': 'Invalid JSON: {}'.format(e)})
    return json.dumps(jobs.add_job(job['start'], job['end']))

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0')

```

```

# jobs.py
q = HotQueue("queue", host='172.17.0.1', port=6379, db=1)
rd = redis.StrictRedis(host='172.17.0.1', port=6379, db=0)

def generate_jid():
    """
    Generate a pseudo-random identifier for a job.
    """
    return str(uuid.uuid4())

def generate_job_key(jid):
    """
    Generate the redis key from the job id to be used when storing, retrieving or updating
    a job in the database.
    """
    return 'job.{}'.format(jid)

def instantiate_job(jid, status, start, end):
    """
    Create the job object description as a python dictionary. Requires the job id,
    status, start and end parameters.
    """
    if type(jid) == str:
        return {'id': jid,
                'status': status,
                'start': start,
                'end': end
               }
    return {'id': jid.decode('utf-8'),
            'status': status.decode('utf-8'),
            'start': start.decode('utf-8'),
            'end': end.decode('utf-8')
           }

def save_job(job_key, job_dict):
    """Save a job object in the Redis database."""
    rd.hset(job_key, job_dict)

def queue_job(jid):
    """Add a job to the redis queue."""

```

(continues on next page)

(continued from previous page)

```

    ....

def add_job(start, end, status="submitted"):
    """Add a job to the redis queue."""
    jid = _generate_jid()
    job_dict = instantiate_job(jid, status, start, end)
    save_job(.....)
    queue_job(.....)
    return job_dict

def update_job_status(jid, status):
    """Update the status of job with job id `jid` to status `status`."""
    job = get_job_by_id(jid)
    if job:
        job['status'] = status
        _save_job(_generate_job_key(jid), job)
    else:
        raise Exception()

```

```

# worker.py
@<...> # fill in
def execute_job(jid):
    """
    Retrieve a job id from the task queue and execute the job.
    Monitors the job to completion and updates the database accordingly.
    """
    # fill in ...
    # the basic steps are:
    # 1) get job id from message and update job status to indicate that the job has
    ↪ started
    # 2) start the analysis job and monitor it to completion.
    # 3) update the job status to indicate that the job has finished.

```

Now that we have placed all of the functions, we can determine which ones should be public and which ones should be private. In general, we want to limit the number of public functions we have. Public functions represent an API for other modules, and the larger the API, the more difficult it will be to make changes in the future.

To this end, we need to determine which functions are called from external modules and which ones are only used locally. We see that `add_job` is called from the `jobs_api` function in `api.py`, so `add_job` should be public. Additionally, while the code isn't explicitly provided, it is clear that the `execute_job` function will need to call `update_job_status`, so that should be public as well. All the other jobs functions are only used internally and can be made private.

Exercise. After placing the functions in the correct files, add the necessary `import` statements.

Solution. Let's start with `api.py`. We know we need to import the `Flask` class to create the `app` object and to use the flask `request` object. We also use the `json` package from the standard library. Finally, we are using our own `jobs` module.

```

# api.py
import json
from flask import Flask, request
import jobs

```

(continues on next page)

(continued from previous page)

```
# rest of the code same as above...
```

For `jobs.py`, there is nothing from our own code to import (which is good since the other modules will be importing from it, but we do need to import the `StrictRedis` and `HotQueue` classes. Also, don't forget the use of the `uuid` module from the standard lib! So, `jobs.py` becomes:

```
# jobs.py
import uuid
from hotqueue import HotQueue
from redis import StrictRedis

# rest of the code same as above...
```

Finally, on the surface it doesn't appear that the worker needs to import anything, but we know it needs the `q` object to get items. It's hidden by the missing decorator. Let's go ahead and import it:

```
# worker.py
from jobs import q

# rest of the code same as above...
```

Take-Home Exercise. Write code to finish the implementations for `_save_job` and `_queue_job`.

Solution. The `_save_job` function should save the job to the database, while the `_queue_job` function should put it on the queue. We know how to write those:

```
def _save_job(job_key, job_dict):
    """Save a job object in the Redis database."""
    rd.hset(job_key, mapping=job_dict)

def _queue_job(jid):
    """Add a job to the redis queue."""
    q.put(jid)
```

Take-Home Exercise. Fix the calls to `_save_job` and `execute_job` within the `add_job` function. *Solution.* The issue in each of these are the missing parameters. The `_save_job` takes `job_key`, `job_dict`, so we just need to pass those in. Similarly, `_queue_job` takes `jid`, so we pass that in. The `add_job` function thus becomes:

```
def add_job(start, end, status="submitted"):
    """Add a job to the redis queue."""
    jid = _generate_jid()
    job_dict = _instantiate_job(jid, status, start, end)
    # update call to save_job:
    save_job(_generate_job_key(jid), job_dict)
    # update call to queue_job:
    queue_job(jid)
    return job_dict
```

Take-Home Exercise. Finish the `execute_job` function. This function needs a decorator (which one?) and it needs a function body.

The function body needs to:

- update the status at the start (to something like “in progress”).

- update the status when finished (to something like “complete”).

For the body, we will use the following (incomplete) simplification:

```
update_job_status(jid, ....)
# todo -- replace with real job.
time.sleep(15)
update_job_status(jid, ....)
```

Solution. As discussed before, we saw in class we can use the `q.worker` decorator to turn the worker into a consumer.

As for `execute_job` itself, we are given the body, we just need to fix the calls to the `update_job_status()` function. The first call puts the job “in progress” while the second sets it to “complete”. So the function becomes:

```
@<...> # fill in
def execute_job(jid):
    update_job_status(jid, "in progress")
    time.sleep(15)
    update_job_status(jid, "complete")
```

Note that we are using the `update_job_status` function from `jobs.py` now, so we need to import it. The final `worker.py` is thus:

```
from jobs import q, update_job_status

@q.worker
def execute_job(jid):
    jobs.update_job_status(jid, 'in progress')
    time.sleep(15)
    jobs.update_job_status(jid, 'complete')
```

Take-Home Exercise. Modify the definition of the `q` and `rd` objects to not use a hard-coded IP address but to instead read the IP address from an environment variable, `REDIS_IP`.

Solution. We can use `os.environ.get("some_string")` to get the value of an environment variable.

```
q = HotQueue("queue", host='172.17.0.1', port=6379, db=1)
rd = redis.StrictRedis(host='172.17.0.1', port=6379, db=0)
```

becomes

```
import os

# read the ip address from the variable REDIS_IP, and provide a default value in case it_
→ is not
# set
redis_ip = os.environ.get('REDIS_IP', '172.17.0.1')
# create the q and rd objects using the variable
q = HotQueue("queue", host=redis_ip, port=6379, db=1)
rd = redis.StrictRedis(host=redis_ip, port=6379, db=0)
```

8.3 Deploying to k8s

In this lecture, we will bring everything together and deploy an updated version of our Flask API system to k8s that includes a Jobs endpoint and a worker deployment. But first, we need to finish the worker. At the end of this module, the student should be able to:

- Complete the basic worker program functionality that can consume messages from the queue and run an associated computation, storing metadata about the job in Redis.
- Convert their Python worker program to a daemon that is always running in the background.
- Containerize the worker program and create a deployment in Kubernetes to deploy the worker to the class cluster.

8.3.1 Daemonizing the Worker

In a Unix-like operating system, a *daemon* is a type of program that runs unobtrusively in the background, rather than under the direct control of a user. The daemon waits to be activated by an occurrence of a specific event or condition.

In summary: A daemon is a long-running background process that answers requests or responds to events.

Recall the high-level architecture of our Jobs API:

- Our Flask API will play the role of producer.
- One or more “worker” programs will play the role of consumer.
- Workers will receive messages about new jobs to execute and performing the analysis steps.
- Workers will oversee the execution of the analysis steps and update the database with the results.

Therefore, our worker program is an example of a daemon that will simply run in the background, waiting for new messages to arrive and executing the corresponding jobs.

We have actually already seen how to turn our Python code into a worker daemon. Let us recall that here:

- We create a new file, `worker.py`, where we put all code related to processing a job.
- The `worker.py` will import a queue object from a `jobs.py` module
- The `worker.py` file includes a function that can take a message from the queue and start processing a job.
- The worker will use the queue object’s `worker` decorator to turn this function into a consumer.
- By adding a call to the function at the bottom of `worker.py`, the worker can be run as a daemon.

Here is a skeleton of the `worker.py` module –

```
# worker.py skeleton
from jobs import q

@q.worker
def do_work(item):
    # do something with item...

do_work()
```

To execute our worker, we simply issue the command `python worker.py` from the command line. Let’s step through what happens, just to make sure this is clear.

1. When `python worker.py` is called from the command line, the python interpreter reads each line of the `worker.py` file and executes any statements it finds in order, from top to bottom.

2. The first line it encounters is the import statement. This imports the definition of `q` from the `jobs.py` file (not included above).
3. Next it hits the decorator and the definition of the function, `do_work(item)`. It checks the syntax of this definition.
4. Finally, it executes the `do_work()` function at the bottom. Since this function is decorated with the `q.worker` decorator, it runs indefinitely, consuming messages from the Redis `q` queue.

8.3.2 Python Buffering

By default, Python buffers output and does not send it to stdout immediately. That has implications for seeing logs using docker or kubernetes. To turn off buffering, use the `-u` flag when calling Python; for example,

```
python -u main.py
```

Another option is to use the `PYTHONUNBUFFERED` environment variable, e.g.,

```
export PYTHONUNBUFFERED=1
```

For the exercise above, we'll use the `-u` option. We'll set this in the Dockerfile for our worker.

8.3.3 Containerizing the Worker

There are multiple ways to containerize the worker, but the simplest approach is to add the `worker.py` code to the same image with the flask API code, and use different commands when running the web server vs running the worker.

For example, the Dockerfile could look like:

```
# Image: jstubbs/animals-service
FROM python:3.9

ADD requirements.txt /requirements.txt
RUN pip install -r requirements.txt
COPY source /app
WORKDIR /app

ENTRYPOINT ["python"]
# note the use of the -u option
COMMAND ["-u", "worker.py"]
```

When running the flask application, the entrypoint and command are already correct. For running the worker, we simply update the command to be `“worker.py”` instead of `“app.py”`.

Exercise. Update your Dockerfile to include an entrypoint and a command that can be used for running both the flask web application and the worker program. Build the new version of your image and push it to Docker Hub.

8.3.4 Deploying to k8s

We're now ready to deploy our complete system to k8s. You should already have deployments and services in k8s for the Flask API and the Redis database, and you should also already have a PVC for Redis to persist state to a volume.

What's left is to add a deployment for the worker pods. Do we need to add a service or PVC for the workers? Why or why not?

Exercise. Create a deployment for your worker pods. Put 2 replicas and be sure to set the command correctly. See above. A deployment skeleton is included below for you to use if you like. Think through the values of each section; some properties/stanzas may not be needed for the worker.

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: <...>
  labels:
    app: <...>
spec:
  replicas: <...>
  selector:
    matchLabels:
      app: <...>
  template:
    metadata:
      labels:
        app: <...>
    spec:
      containers:
        - name: <...>
          imagePullPolicy: Always
          image: <...>
          command: <...>
          env:
            - <...>
          ports:
            - <...>
```

8.3.5 Code Repository

It is good to keep your code and deployment files organized in a single repository. Consider using a layout similar to the following:

```
deploy/
  api/
    deployment.yml
    service.yml
  db/
    deployment.yml
    pvc.yml
    service.yml
  worker/
```

(continues on next page)

(continued from previous page)

```
deployment.yml
Dockerfile
source/
  api.py
  jobs.py
  worker.py
```

8.4 Diagrams for Software Design

In this module, we provide a brief introduction to diagrams for software design. By the end of this module, the student should be able to:

- Explain the basic utility of diagrams for software design
- Describe the differences between structural diagrams and behavioral diagrams.
- Create different types of structural and/or behavioral diagrams for their flask-based API system.

As a software system grows in its size and complexity, it becomes increasingly difficult to describe in written text. Diagrams help us visualize the components of a software system, allowing us to provide a more concise and holistic description. There are many tools and approaches to creating diagrams for software systems. In this short module, we provide only a brief introduction to diagrams for software design, and we include some references for further reading.

Generally speaking, a diagram can convey two types of information:

- Static relationships between components, sometimes called a structural diagram.
- Dynamic information describing how the system changes or how different components react to each other. These are called behavior diagrams.

8.4.1 Structural Diagrams

With a structural diagram, the goal is to represent the components of the system with different shapes, and to indicate connections between the components, usually with an arrow. Different diagrams will represent different types of components in the system at different granularity. For example, we could use boxes to represent:

- Classes in software utilizing object oriented programming
- Modules and libraries in a large code base
- Containers or services in a distributed system utilizing a microservice architecture

Similarly, arrows between components represent different types of connections.

- For classes in OOP, arrows represent different types of relationships between the classes, such as inheritance (when one class is a child of another class) or dependency (when one class uses methods from another class).
- For a diagram containing microservices, arrows represent the communication between services.

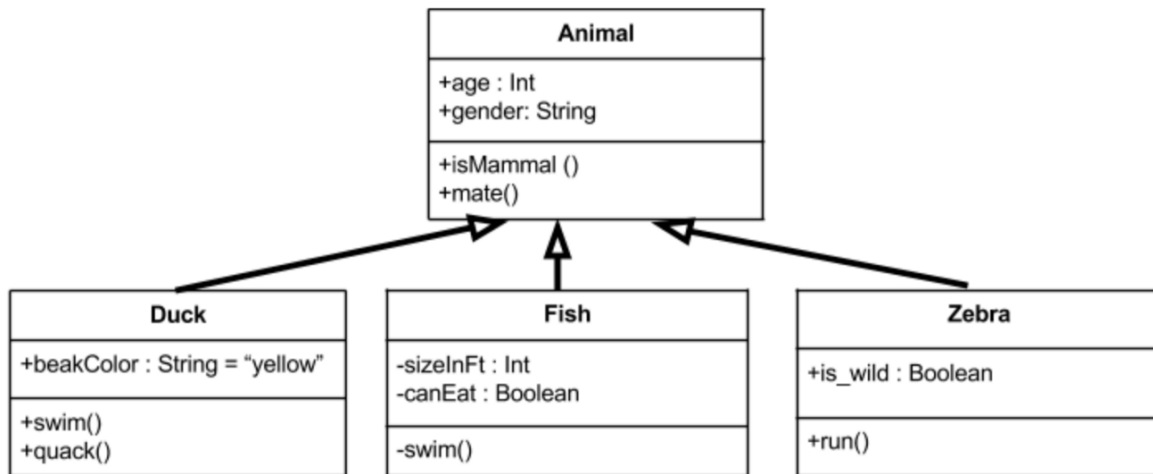


Fig. 1: An example of a class diagram

Kubernetes - Deployment and Container Orchestration

PVC - persistent storage (Ceph)

Service - Networking

Deployments

- API Server
- Persistence
- Async components

Jobs

- Initializations
- DB Migrations

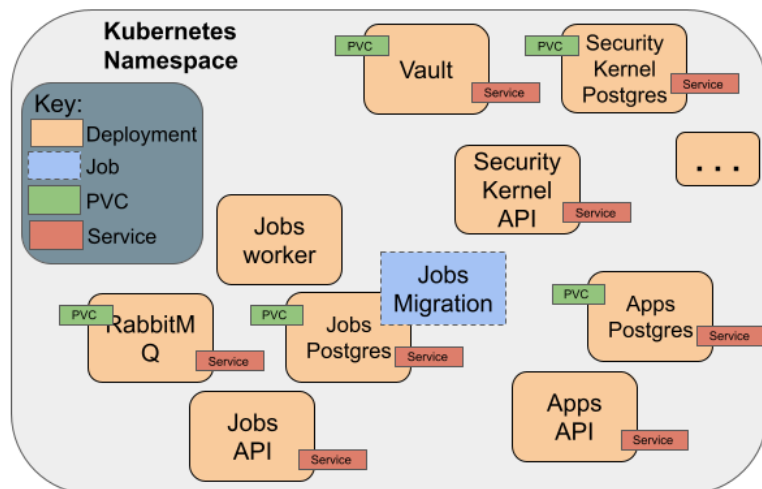


Fig. 2: An example of a microservices diagram

8.4.2 Behavioral Diagrams

By contrast to structural diagrams, behavioral diagrams capture the how components within the system respond to changes. Some common behavioral diagrams include:

Flowcharts

Flowcharts depict the step-by-step process of an algorithm or component of a program. Each box represents a different step in the process. Different shapes are used for different types of steps, such as a rhombus for collecting input, a diamond for an IF/THEN conditional, and a rectangle for a computation.

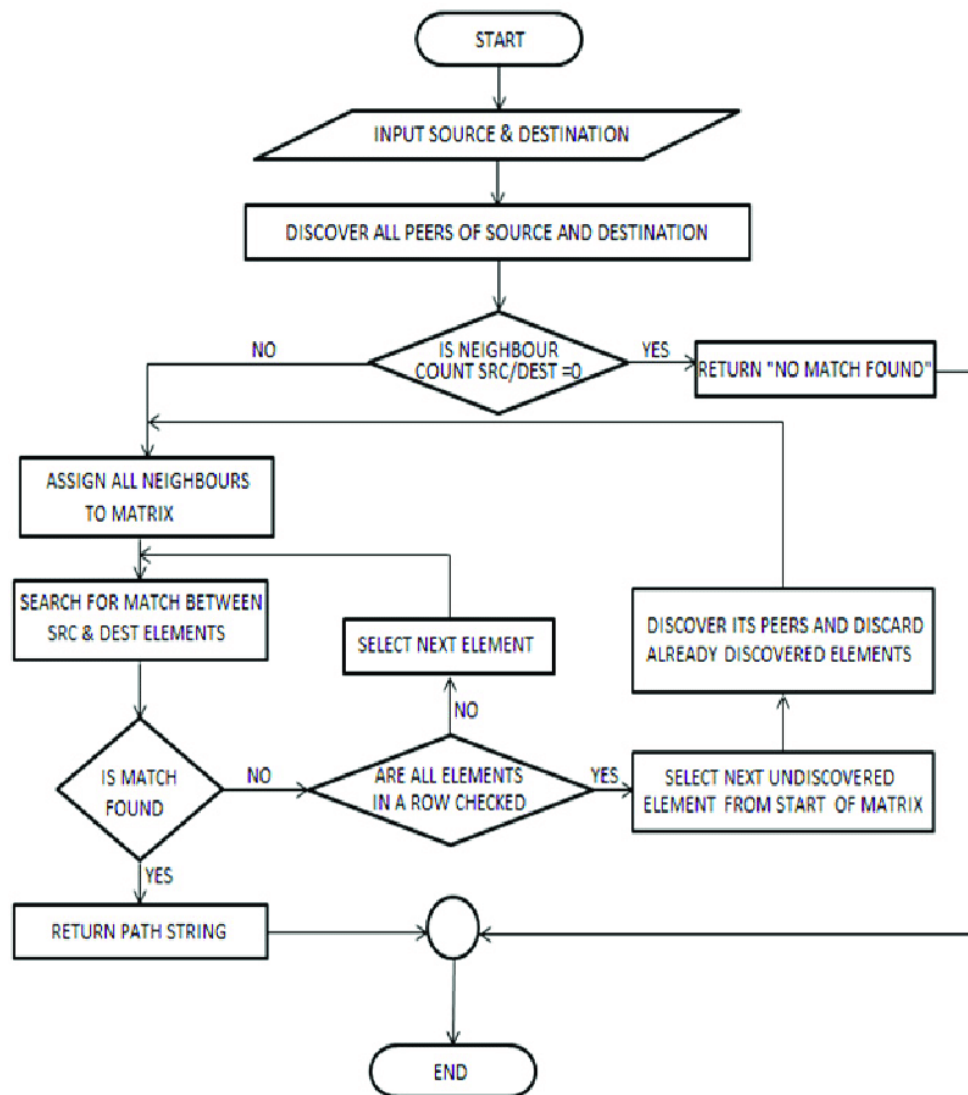


Fig. 3: A flowchart of the all-pairs-shortest-path (APSP) algorithm.

Sequence Diagrams

Sequence diagrams describe the sequence of interactions (usually, communications) between components of a system and even end users. The components represented in a sequence diagram could be web pages or URLs within a single web application, or they could be entirely different services in a microservice architecture.

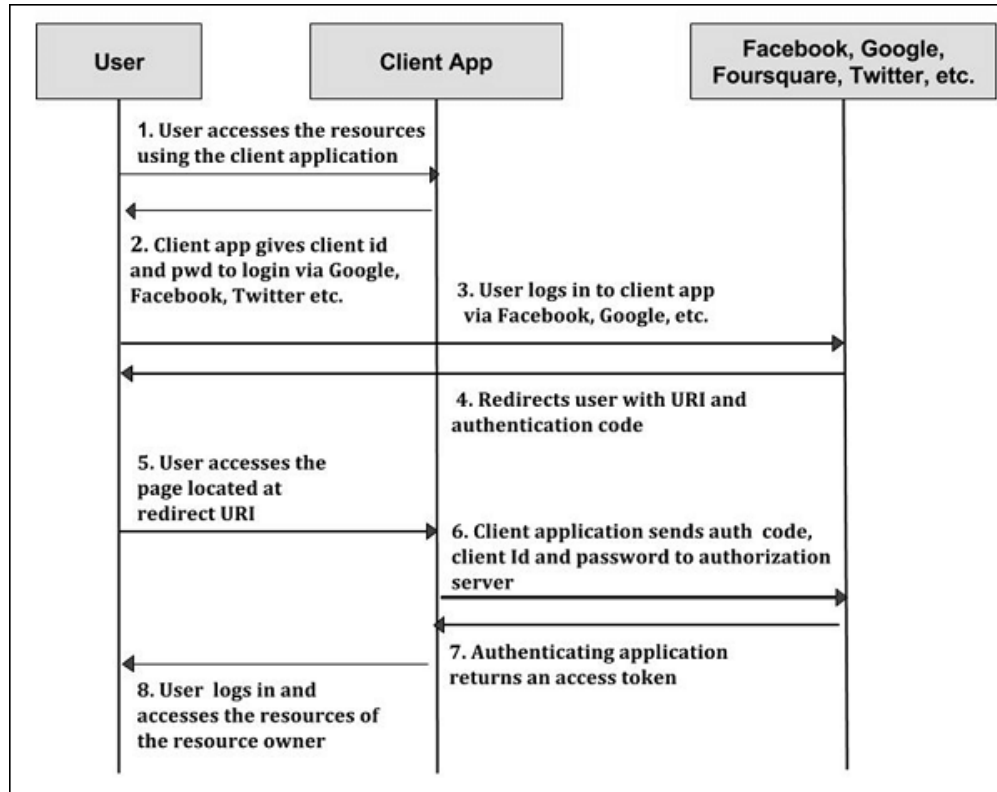


Fig. 4: A sequence diagram of the OAuth2 authorization code flow.

8.4.3 Unified Modeling Language (UML)

The Unified Modeling Language (UML) is a modeling language for describing diagrams. It was created in 1994 and became an ISO standard in 2005. There are two major versions of UML, v1 and v2. In v2, there are at least 14 different types of diagrams.

Some UML diagrams, such as the class diagram, are precise enough to be able to generate code from them. While this is a neat idea, in practice some software engineers find UML heavyweight and cumbersome. If you are interested in UML, there are a number of tutorials on the web.

Here is an example description of a C4 UML Plant diagram describing a hypothetical COE 332 final project:

```

@startuml
!include C4_Container.puml

LAYOUT_TOP_DOWN()
LAYOUT_WITH_LEGEND()

title System Diagram for Example COE 332 Final Project
  
```

(continues on next page)

(continued from previous page)

```

Person(scientist, Scientist, "A scientist interested in studying sun spots")

System_Boundary(c1, "Sun Spots Analysis Platform") {
    Container(api, "RESTful API", "Python, Flask", "Provides endpoints for retrieving
↪data and launching analysis jobs.")
    Container(worker, "Worker", "Python worker, docker container", "Receives jobs from
↪the task queue, and works them to completion.")
    ContainerDb(database, "Redis", "Redis Database", "Stores sun spots dataset and
↪information and jobs in the system.")
    Container(task_queue, "Task Queue", "Redis Task Queue", "Stores pending jobs in the
↪system")
}
Rel_Down(scientist, api, "Uses", "HTTP/S")

Rel_Down(task_queue, worker, "Dequeues jobs")
Rel_Down(database, worker, "Reads from and writes to")
Rel_Down(api, database, "Reads from and writes to", "python, redis")
Rel_Down(api, task_queue, "Adds new jobs to queue")

@enduml

```

This generates the following diagram:

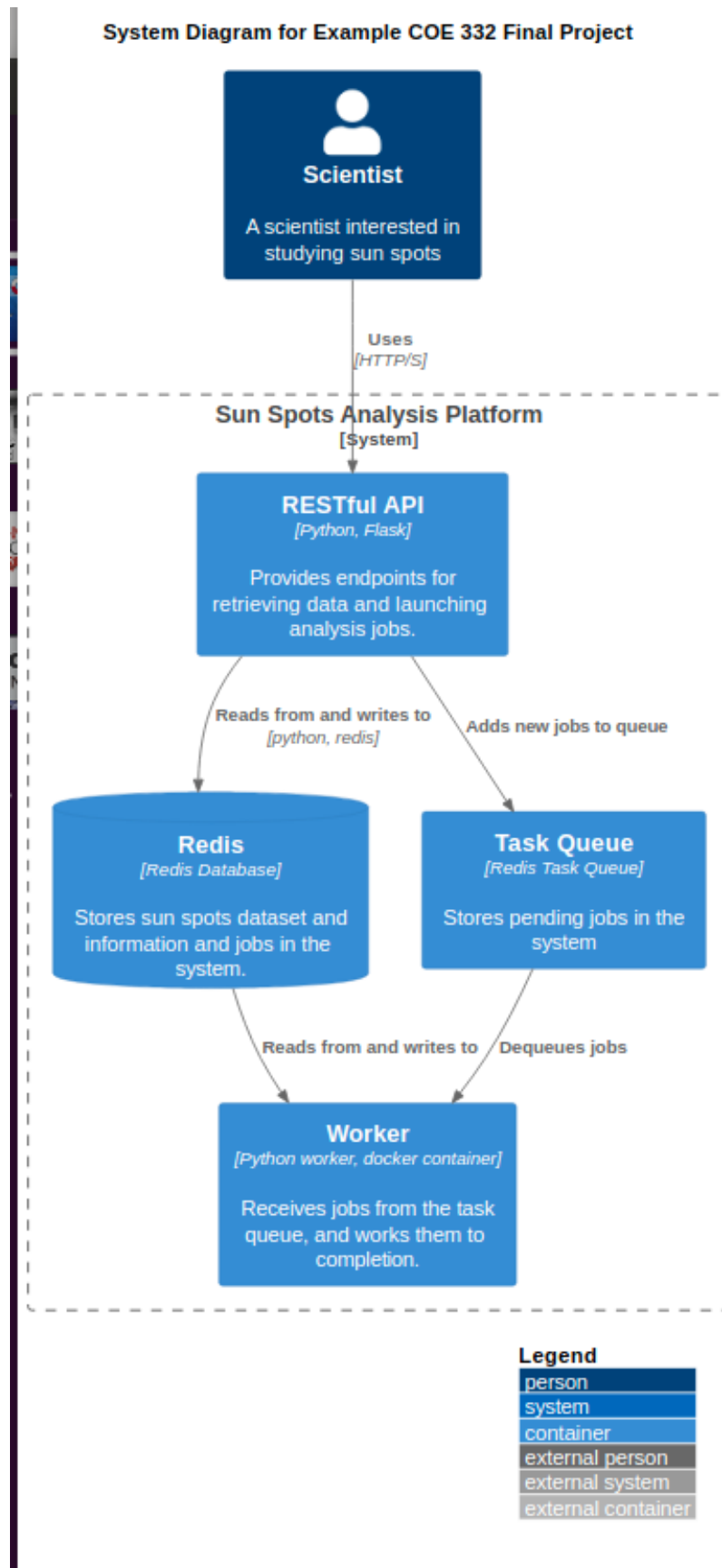
8.4.4 Options for Creating Diagrams

There are a lot of options for creating diagrams:

1. Google Slides/Google Draw - Both options allow you to create basic shapes and connectors, fill, arranges, etc. Free with a google account.
2. draw.io - Similar to google slides but some find it to be more ergonomic to use. Free.
3. Microsoft Powerpoint - Similar to the other options above; requires access to Microsoft Office.

And if you want to make diagrams from UML...

4. Visual Paradigm - This is kind of like the others above where you click and drag boxes and arrows (I think). I got the "Community Edition" installed on Ubuntu without much trouble. It is free.
5. kroki.io - This project is kind of fun. It provides an HTTP API for making diagrams (what could be better?) You describe your diagrams in text (for example, in UML) and make an HTTP request to the diagram endpoint and it returns to you HTML that renders your figure. You can use their community server, or you can run the whole API as a docker container on your machine. It's all free.



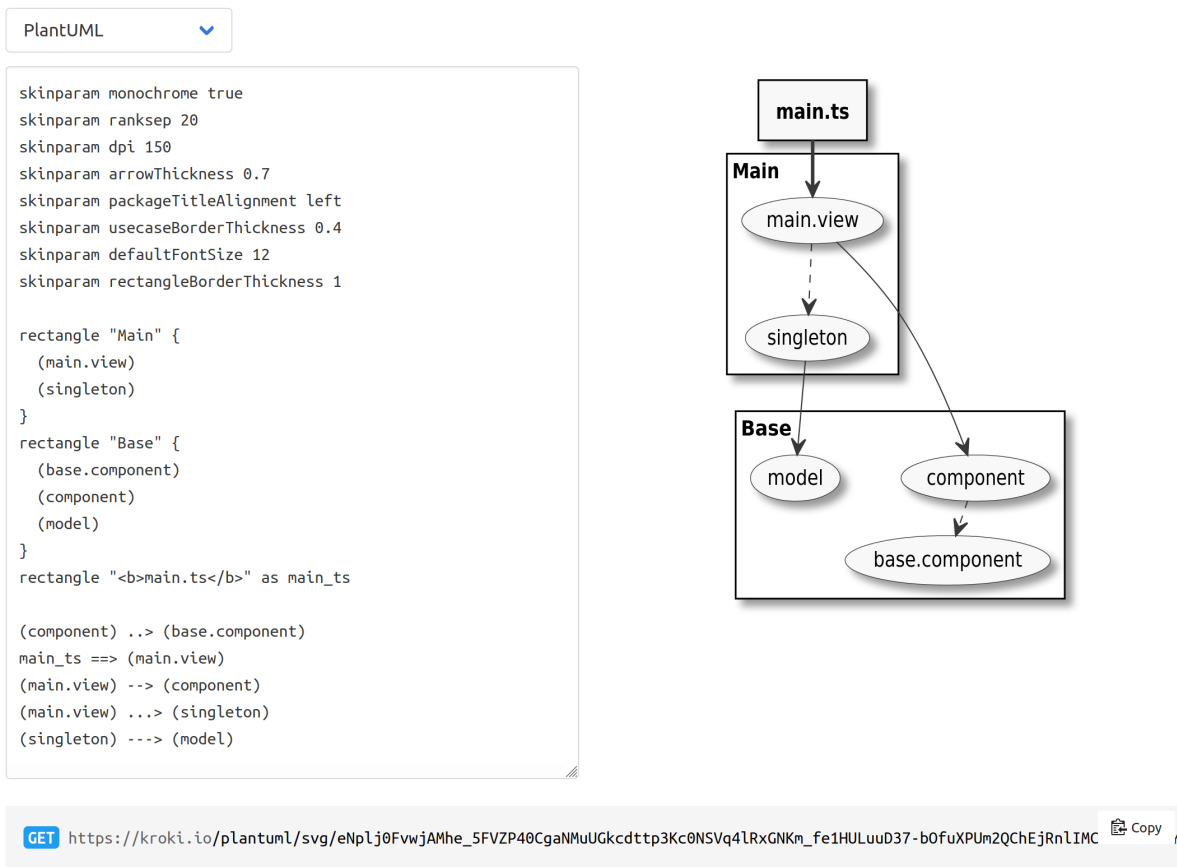


Fig. 6: From the kroki.io documentation.

UNIT 9: INTEGRATION TESTING, CONTINUOUS INTEGRATION

Up to this point, we have worked on all of the different tools, technologies, and components that will go into the software systems we will build for our final projects. In this unit, we will further discuss how we will approach building our software systems between our development environment (ISP02) and the deployment environment (kube-2). We will also learn some new techniques to help speed up the development cycle and catch errors as we go - namely integration testing and continuous integration.

9.1 Development Environment

Development of the software systems for our final projects should be performed on the ISP server. The containers built and run on ISP02 will be ephemeral - only lasting long enough to test and debug new features. They should never be seen by end users. After going through this module, students should be able to:

- Set up the files and folders necessary to work on the final project
- Describe each component and why they are important to the overall system
- Describe the general development cycle from a new idea all the way to production
- Write targets for the major steps into a Makefile

9.1.1 File Organization

Our final software systems should have three main components:


1. A Flask API front end for accessing data and jobs functionality
2. A Redis database for storing job and queue information
3. A Worker back end which runs the analysis job

An example file organization scheme for developing this API may look like:

```
my-api/  
├── data  
│   └── dump.rdb  
├── docker  
│   ├── Dockerfile.api  
│   └── Dockerfile.wrk  
├── Makefile  
├── README.md  
├── src  
│   └── api.py
```

(continues on next page)

(continued from previous page)



```
├─ jobs.py
└─ worker.py
```

In this example, the `data/` subfolder is mounted inside a Redis container. All the data is captured in regular intervals in `dump.rdb` for testing purposes only (real end users will not see this copy of the data).

The `docker/` subfolder contains a Dockerfile for each service. We don't need a Dockerfile for the redis container because we will use the stock `redis:6`.

The `Makefile` will be an essential part of our development cycle (revisited below).

The `src/` folder contains the source Python scripts that are injected into the API and worker containers. This is where the majority of the development will occur.

Tip: Make sure to put your folders under version control and commit regularly as you work. Don't save it all until the end.

QUESTIONS

- Is the data set I'm working with part of this repository? Why or why not?
- Should we commit `dump.rdb` to GitHub? Why or why not?
- I am not using a `requirements.txt` file above. Should I?
- I am using two Dockerfiles - one for the worker and one for the API. Should I be doing this or should I just have one Dockerfile?

9.1.2 Docker

We previously talked at great length about why it is a good idea to containerize an app / service that you develop. One of the main reasons was for portability to other machines. All the development / testing done in this development environment will directly translate to our deployment environment (Kubernetes), as we will see in the next module.

The development cycle for each of the three containerized components generally follows the form:

1. Edit some source code (e.g. add a new Flask route)
2. Delete any running container with the old source code
3. Re-build the image with `docker build`
4. Start up a new container with `docker run`
5. Test the new code / functionality
6. Repeat

This 6-step cycle is great for iterating on the API and Worker containers independently, or at the same time. However, watch out for potential error sources. For example if you take down the Redis container, a worker container that is in the middle of watching the queue may also go down and will need to be restarted (once a new Redis container is up).

9.1.3 Makefile

We previously introduced Makefiles as a useful automation tool for testing your services. Now that our development cycle is becoming longer and more complicated, it is worth revisiting. Here, we will set up a Makefile to help with the 6-step cycle above. Using certain keywords (called “targets”) we will create shortcuts to cleaning up running containers, re-building docker images, and running new containers.

Targets are listed in a file called `Makefile` in this format:

```
target: prerequisite(s)
      recipe
```

Targets are short keywords, and recipes are shell commands. For example, a simple target might look like:

```
ps-me:
      docker ps -a | grep wallen
```

Put this text in a file called `Makefile` in your current directory, and then you simply need to type:

```
[isp02]$ make ps-me
```

And that will list all the docker containers with the username ‘wallen’ either in the image name or the container name. Makefiles can be further abstracted with variables to make them a little bit more flexible. Consider the following Makefile:

```
NAME ?= wallen

all: ps-me im-me

im-me:
      docker images | grep ${NAME}

ps-me:
      docker ps -a | grep ${NAME}
```

Here we have added a variable `NAME` at the top so we can easily customize the targets below. We have also added two new targets: `im-me` which lists images, and `all` which does not contain any recipes, but does contain two prerequisites - the other two targets. So these two are equivalent:

```
# make all targets
[isp02]$ make all

# or make them one-by-one
[isp02]$ make ps-me
[isp02]$ make im-me

# Try this out:
[isp02]$ NAME="redis" make all
```

EXERCISE

Write a Makefile that, at a minimum:

1. Builds all necessary images for your app from Dockerfile(s)
2. Starts up new containers / services
3. Removes running containers in your namespace (be careful!)

9.2 Deployment Environment

Our deployment environment for this API will be the class Kubernetes cluster. It could just as easily be AWS, or Azure, or Google Cloud, or another Kubernetes cluster. Remember if you containerize everything, it becomes extremely portable. In contrast to our development environment, the Kubernetes deployment is meant to be long-lasting, highly available, and consumable by the public. We will have *test* and *prod* deployments, so that new changes can be seen by developers in the *test* deployment environment (sometimes also called “staging”) before finally making their way to the *prod* (production) deployment environment. After going through this module, students should be able to:

- Name and organize YAML files for test and prod deployments of a software system
- Sync files between two remote copies of a Git repository
- Version code following semantic versioning specification
- Attach a NodePort service to an API deployment
- Deploy production and test copies of software system in Kubernetes

9.2.1 File Organization

To support the deployment environment (prod and test), our file tree may grow to something similar to the following:

```
my-api/
├── data
│   └── dump.rdb
├── docker
│   ├── Dockerfile.api
│   └── Dockerfile.wrk
├── kubernetes
│   ├── prod
│   │   ├── api-deployment.yml
│   │   ├── api-service.yml
│   │   ├── db-deployment.yml
│   │   ├── db-pvc.yml
│   │   ├── db-service.yml
│   │   └── wrk-deployment.yml
│   └── test
│       ├── api-deployment.yml
│       ├── api-service.yml
│       ├── db-deployment.yml
│       ├── db-pvc.yml
│       ├── db-service.yml
│       └── wrk-deployment.yml
└── Makefile
```

(continues on next page)

(continued from previous page)

```

├── README.md
├── src
│   ├── api.py
│   ├── jobs.py
│   └── worker.py

```

In this example, you will find 12 new YAML files with somewhat descriptive names. Six are organized into a ‘test’ directory, and six are organized into a ‘prod’ directory, although there are other acceptable ways to organize these files.

These YAML files closely follow the naming convention and content we have seen in previous lectures. There are three deployments - one each for the API, database, and worker. There are two services - one each for the API and database. And, there is a persistent volume claim (PVC) for the database. Without looking into the contents of the files, it is clear what function each serves, and it is clear that there is one-to-one correspondence between the test and prod deployments.

9.2.2 Test Deployment

The purpose of this testing / staging environment is to see the entire API exactly as it appears in production before actually putting new code changes into production.

Generally the process to get code into testing follows these steps:

1. Develop / test code in the development environment (ISP) as described in the previous module
2. Push code to GitHub and tag it with an appropriate version number (avoid using “latest” - see Versioning section below)
3. Push images to Docker Hub - Kubernetes needs to pull from here. Make sure the Docker image tag matches the GitHub tag so you always know what exact version of code is running.
4. Edit the appropriate testing deployment(s) with the new tags and apply the changes. Pods running within a deployment under the old tag number should be automatically terminated.

The yaml files above can be applied one by one, or the entire directory at a time like the following:

```
[kube-2]$ kubectl apply -f kubernetes/test/
```

Kubernetes will apply all the files found in the test folder. Be careful, however, about the order in which things are applied. For example, the Redis DB deployment needs the PVC to exist in order to deploy successfully. But, Kubernetes is usually pretty smart about this kind of thing, so it should keep retrying all deployments, services, and PVCs until everything is happy and connected.

Once deployed, you should rigorously test all services using the Python debug pod and, if applicable, the NodePort Service connection to the outside world. We will see more on automating integration tests later in this unit.

9.2.3 Syncing Git Repos

A challenge you might encounter while working between the class VM (ISP) and the Kubernetes cluster (kube-2) is keeping your Git repository in sync. If you are doing most of your development ISP, you will need to commit and push those files to GitHub, then clone them on to kube-2. In practice, we will likely be editing different sets of files on the two machines (YAML files on kube-2, everything else on ISP), but it is a good idea to stay organized and keep everything in sync. A sample workflow may resemble:

- 1) Imagine the repository on GitHub is newly cloned to ISP and kube-2, and everything is in sync. Then, edit some files on ISP, commit changes, and push to GitHub:

```
[isp02]$ git add .  
[isp02]$ git commit -m "message"  
[isp02]$ git push
```

- 2) Now we would say GitHub is one commit ahead of the remote repository on kube-2. Log in to kube-2, and pull in the remote changes:

```
[kube-2]$ git remote update  
[kube-2]$ git pull
```

- 3) Everything is back in sync again. Next, edit some files on kube-2, commit changes, and push to GitHub:

```
[kube-2]$ git add .  
[kube-2]$ git commit -m "message"  
[kube-2]$ git push
```

- 4) Now we would say GitHub is one commit ahead of the report repository on ISP. Log back in to ISP, and pull in the remote changes:

```
[isp02]$ git remote update  
[isp02]$ git pull
```

Finally the origin (GitHub) and both remote copies of the repository are in sync again. This strategy works not only for keeping copies of your own repository in sync between multiple machines, but it also works for collaborating with others on a repository. (Although there are [better ways to collaborate](#)).

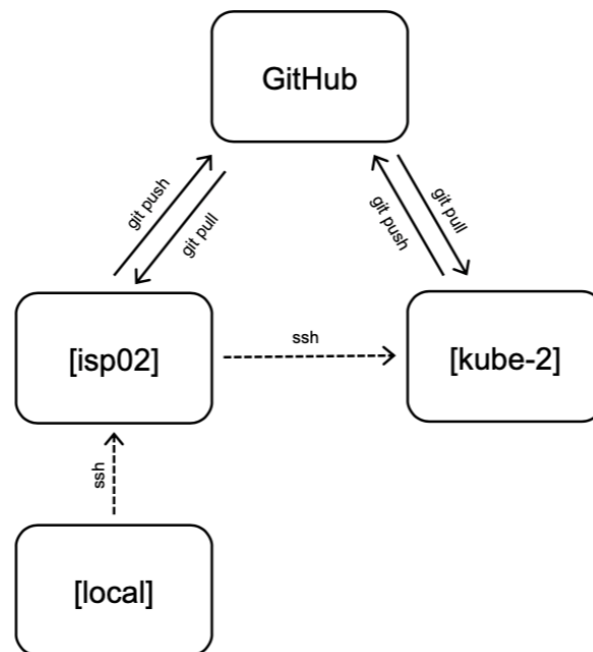


Fig. 1: Remember to keep remote repos synced with GitHub.

9.2.4 Versioning

We have not spent much time discussing versioning in this class other than to see do not use the tag 'latest' when versioning your repos or Docker images. There is a well-accepted standard for versioning called 'Semantic Versioning'. It follows the specification:

Given a version number **MAJOR.MINOR.PATCH**, increment the:

- **MAJOR** version when you make incompatible API changes,
- **MINOR** version when you add functionality in a backwards compatible manner, and
- **PATCH** version when you make backwards compatible bug fixes.

You can assign a tag to the current state of a repository on the command line by doing:

```
[isp02]$ git tag -a 0.1.0 -m "first release"
[isp02]$ git push origin 0.1.0
```

Tip: Do you have a new software system that just kind of works and has a little bit of functionality, but you don't know what version tag to assign it? A good place to start is version 0.1.0.

9.2.5 NodePort Service

So far we have only been able to interact with our deployed APIs via another pod that is running on the Kubernetes cluster. We can also reach our deployed APIs from the outside world (public URLs) using a NodePort Service.

In Kubernetes, a range of ports, called NodePorts, are open on every node of the cluster. We have assigned two unique ports to each student (for test and prod), and proxied those ports to two public URLs, exposing your test and prod APIs to the outside world.

On the Kubernetes cluster, we created a file called `portinfo` in each student's home directories. Cat the file to see the contents (every student's file is slightly different):

```
[kube-2]$ cat ~/portinfo
docker port: 5042
kube port 1: 30042
kube port 2: 30142
public url 1: "https://isp-proxy.tacc.utexas.edu/USERNAME-1/"
public url 2: "https://isp-proxy.tacc.utexas.edu/USERNAME-2/"
```

The important info from the example above is that using NodePort `30042` will expose an interface to the API at public URL `https://isp-proxy.tacc.utexas.edu/USERNAME-1/`, and using NodePort `30142` will expose an interface to the API at public URL `https://isp-proxy.tacc.utexas.edu/USERNAME-2/`. In practice, one of these ports should be attached to your test deployment, and the other should be attached to your production deployment.

An example NodePort service YAML file may include (be sure to use an app selector that matches the correct deployment):

```
---
apiVersion: v1
kind: Service
metadata:
  name: hello-service
spec:
```

(continues on next page)

(continued from previous page)

```
type: NodePort
selector:
  app: hello-app
ports:
- name: hello-app
  port: 5000
  targetPort: 5000
  nodePort: 30042
```

9.2.6 Production Deployment

If everything with the test / staging deployment looks good and passes tests, follow the same steps for your production environment. Kubernetes is fast at stopping / starting containers, and the services should provide pretty seamless access to the underlying API. If larger-scale changes are needed and significant downtime is anticipated, it would be a good idea to post an outage notice to users.

9.2.7 Additional Resources

- [Collaborate on Git Repos](#)
- [Semantic Versioning](#)

9.3 Integration Testing

Unlike unit tests, integration tests exercise multiple components, functions, or units of a software system at once. Some properties of integration tests include:

- Each test targets a higher-level capability, requirement or behavior of the system, and exercises multiple components of the system working together.
- Broader scope means fewer tests are required to cover the entire application/system.
- A given test failure provides more limited information as to the root cause.

It's worth pointing out that our definition of integration test leaves some ambiguity. You will also see the term “functional tests” used for tests that exercise entire aspects of a software system. After going through this module, students should be able to:

- Identify Python frameworks for integration testing
- Identify aspects of a software system that should be tested with integration testing
- Use the Python requests library to interact with the API of your software system
- Write and execute useful integration tests using `pytest` and `assert` statements

9.3.1 Challenges When Writing Integration Tests

Integration tests against large, distributed systems with lots of components that interact face some challenges.

- We want to keep tests independent so that a single test can be run without its result depending on other tests.
- Most interesting applications change “state” in some way over time; e.g., files are saved/updated, database records are written, queue systems updated. In order to properly test the system, specific state must be established before and after a test (for example, inserting a record into a database before testing the “update” function).
- Some components have external interactions, such as an email server, a component that makes an update in an external system (e.g. GitHub) etc. A decision has to be made about whether or not this functionality will be validated in the test and if so, how.

9.3.2 Initial Integration Tests for Our Flask API

For our first set of integration tests, we’ll use the following strategy:

- Start the Flask API, Redis DB, and Worker services
- Use `pytest` and `requests` to make requests directly to the running API server
- Check various aspects of the response; each check can be done with a simple assert statement, just like for unit tests

9.3.3 A Simple `pytest` Example

Similar to unit tests, we will use `assert` statements to check that some input data or command returns the expected result. A simple example of using `pytest` might look like:

```

1 import pytest, requests
2
3 def test_flask():
4     response = requests.get('http://localhost:5000/route')
5     assert response.status_code == 200

```

This small test just checks to make sure curling the route (with the Python requests library) returns a successful status code, 200.

As we have seen before, test scripts should be named strategically and organized into a subdirectory similar to:

```

my-api/
├── data
│   └── dump.rdb
├── docker
│   ├── Dockerfile.api
│   └── Dockerfile.wrk
├── Makefile
├── README.md
├── src
│   ├── api.py
│   ├── jobs.py
│   └── worker.py
└── test
    └── test_api.py

```

Run the test simply by typing this in the top (`my-api/`) directory:

```
[isp02]$ pytest
===== test session starts =====
platform linux -- Python 3.6.8, pytest-6.2.3, py-1.10.0, pluggy-0.13.1
rootdir: /home/wallen/coe-332/my-api
collected 1 item

test/test_api.py .                                [100%]

===== 1 passed in 0.17s =====
```

Tip: Don't forget to `pip3 install --user pytest` first.

EXERCISE

Continue working in the test file, `test_api.py`, and write a new functional test that use the `requests` library to make a GET request to the `/jobs` endpoint and check the response for, e.g.:

- The response returns a 200 status code
- The response returns a valid JSON string
- The response can be decoded to a Python dictionary
- Each element of the decoded list is a Python dictionary
- Each dictionary in the result has two keys
- Verify that the type of each key's value is correct

Remember, your services should be running and as much as possible, functional tests should be testing the end-to-end functionality of your entire app.

9.4 Continuous Integration

In a multi-developer environment, typically no one person has complete knowledge of the entire system, and multiple changes can be happening at the same time. Even if the changes are made in different components, it is possible for something to break when they are integrated. The primary goal of Continuous Integration (CI) is to enable multiple developers to work on the same code base while ensuring the quality of the final product. After going through this module, students should be able to:

- Identify the importance of CI to a large software system
- Choose a CI service that meets the needs of their software system
- Perform a integration testing CI workflow with GitHub Actions
- Perform a docker build / docker push CI workflow with GitHub Actions

9.4.1 An Example CI Workflow

An “integration server” (or “build server”) is a dedicated server (or VM) that prepares software for release. The server automates common tasks, including:

- Building software binaries from source code (for compiled languages)
- Running tests
- Creating images, installers, or other artifacts
- Deploying/installing the software

We are ultimately aiming for the following “Continuous Integration” work flow or process; this mirrors the process used by a number of teams working on “large” software systems, both in academia and industry:

- Developers (i.e., you) check out code onto a machine where they will do their work. This could be a VM somewhere or their local laptop.
- They make changes to the code to add a feature or fix a bug.
- Once their work is done they add any additional tests as needed and then run all unit tests “locally” (i.e., on the same machine).
- Assuming the tests pass, the developer commits their changes and pushes to the origin (in this case, GitHub).
- A pre-established build server gets a message from the origin that a new commit was pushed.
- The build server:
 - Checks out the latest version
 - Executes any build steps to create the software
 - Runs unit tests
 - Starts an instance of the system
 - Runs integration tests
 - Deploys the software to a staging environment

If any one of the steps above fails, the process stops. In such a situation, the code defect should be addressed as soon as possible.

9.4.2 Popular Automated CI Services

Jenkins is one of the most popular free open-source CI services. It is server-based, and it requires a web server to operate on.

- Local application
- Completely free
- Deep workflow customization
- Intuitive web interface management
- Can be distributed across multiple machines / VMs
- Rich in features and plugins
- Easy installation thanks to the pre-installed OS X, Unix and Windows packages
- A well-established product with an excellent reputation

TravisCI is another CI service with limited features in the free tier, and a comprehensive paid tier. It is a cloud-hosted service, so there is no need for you to host your own server.

- Quick setup
- Live build views
- Pull request support
- Multiple languages and platforms support
- Pre-installed database services
- Auto deployments on passing builds
- Parallel testing (paid tier)
- Scaling capacity on demand (paid tier)
- Clean VMs for every build
- Mac, Linux, and iOS support
- Connect with Github, Bitbucket and more

GitHub Actions is a relatively new CI service used to automate, customize, and execute software development workflows right in your GitHub repository.

- One interface for both your source code repositories and your CI/CD pipelines
- Catalog of available Actions you can utilize without reinventing the wheel
- Hosted services are subject to usage limits, although the free-tier limits are **fairly generous** (for now)
- Simple YAML descriptions of workflows, many templates and examples available
- It is a newer platform, so not as many features as some of the others, but it is quickly gaining steam

9.4.3 What Will We Do With CI?

Two obvious and useful forms of CI we can incorporate into the development of our final projects with GitHub Actions include:

- 1) Automatically run our integration tests (with pytest) each time new code is pushed to GitHub
- 2) Automatically build a Docker image and push it to Docker Hub each time our code is tagged with a new release

9.4.4 Integration Testing with GitHub Actions

To set up GitHub Actions in an existing repository, create a new folder as follows:

```
[isp02]$ mkdir -p .github/workflows/
```

Within that folder we will put YAML files describing when, how, and what workflows should be triggered. For instance, create a new YAML file (`.github/workflows/integration-test.yml`) to perform our integration testing with the following contents:

```
name: Integration tests with pytest
on: [push]

jobs:
```

(continues on next page)

(continued from previous page)

```

integration-tests-with-pytest:
  runs-on: ubuntu-latest

  steps:
    - name: Check out repo
      uses: actions/checkout@v3

    - name: Stage the data
      run: wget https://raw.githubusercontent.com/wjallen/coe332-sample-data/main/ML_
↪Data_Sample.json

    - name: Create docker bridge network
      run: docker network create API-DB-WRK

    - name: Set up a database
      run: |
        mkdir ./data/
        docker run --name redis-db --network API-DB-WRK -p 6441:6379 -d -v ${PWD}/data:/
↪data redis:6 --save 1 1

    - name: Build and run the API
      run: |
        docker build -f docker/Dockerfile.api -t api:test .
        docker run --name api-test --network API-DB-WRK -p 5041:5000 -d --env REDIS_IP=$
↪{RIP} api:test
      env:
        RIP: redis-db

    - name: Build and run the worker
      run: |
        docker build -f docker/Dockerfile.wrk -t wrk:test .
        docker run --name wrk-test --network API-DB-WRK -d --env REDIS_IP=${RIP} wrk:test
      env:
        RIP: redis-db

    - name: Set up Python
      uses: actions/setup-python@v3
      with:
        python-version: "3.9"

    - name: Install dependencies
      run: pip3 install pytest==7.0.0 requests==2.27.1

    - name: Run pytest
      run: pytest

    - name: Stop images
      run: |
        docker stop wrk-test && docker rm -f wrk test
        docker stop api-test && docker rm -f api-test
        docker stop redis-db && docker rm -f redis-db
        docker network rm API-DB-WRK

```

The workflow above runs our integration tests, and it is triggered on every push (on: [push]). This particular workflow will run in an ubuntu-latest VM, and it has 10 total steps.

Some steps contain a uses keyword, which utilizes a pre-canned action from the catalog of GitHub Actions. For example, the pre-canned actions might be used to clone your whole repository or install Python3. The other steps contain a run keyword which are the commands to run on the VM. In the above example, commands are run to stage the data, set up containers, and run pytest.

QUESTION

In the above example, Python v3.9 and external libraries (pytest, requests) are installed in different steps. Can this be done in one step? Is there a better way to do it?

Trigger the Integration

To trigger this integration, simply edit some source code, commit the changes, and push to GitHub.

```
[isp02]$ git add *
[isp02]$ git commit -m "added a new route to do something"
[isp02]$ git push
```

Then navigate to the repo on GitHub and click the ‘Actions’ tab to watch the progress of the Action. You can click on your saved workflows to narrow the view, or click on a specific instance of a workflow (a “run”) to see the logs.

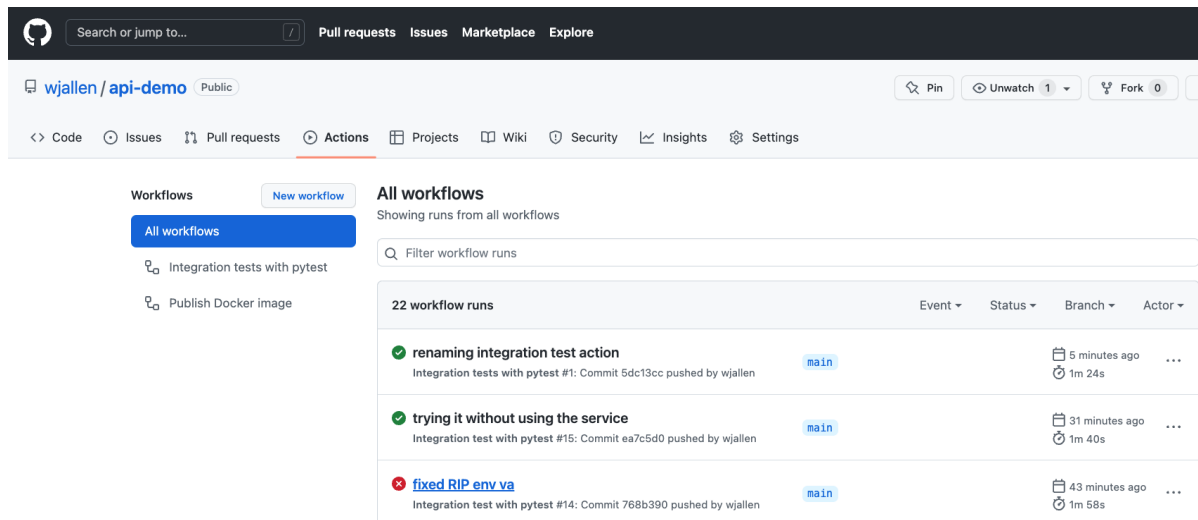


Fig. 2: History of all workflow runs.

By looking through the history of recent workflow runs, you can see that each is assigned to a specific commit and commit message. That way, you know who to credit or blame for successful or errant runs.

9.4.5 Docker Hub Integration with GitHub Actions

Rather than commit to GitHub AND push to Docker Hub each time you want to release a new version of code, you can set up an integration between the two services that automates it. The key benefit is you only have to commit to one place (GitHub), and you can be sure the image on Docker Hub will always be in sync.

Consider the following workflow, located in `.github/workflows/push-to-registry.yml`:

```
name: Publish Docker image

on:
  push:
    tags:
      - '*'

jobs:
  push-to-registry:
    name: Push Docker image to Docker Hub
    runs-on: ubuntu-latest

    steps:
      - name: Check out the repo
        uses: actions/checkout@v3

      - name: Stage the data
        run: wget https://raw.githubusercontent.com/wjallen/coe332-sample-data/main/ML_
↪Data_Sample.json

      - name: Log in to Docker Hub
        uses: docker/login-action@f054a8b539a109f9f41c372932f1ae047eff08c9
        with:
          username: ${ secrets.DOCKERHUB_USERNAME }
          password: ${ secrets.DOCKERHUB_PASSWORD }

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v1

      - name: Extract metadata (tags, labels) for Docker
        id: meta-api
        uses: docker/metadata-action@98669ae865ea3cfffcbbaa878cf57c20bbf1c6c38
        with:
          images: wjallen/mldata-api

      - name: Build and push Docker image
        uses: docker/build-push-action@ad44023a93711e3deb337508980b4b5e9bcdc5dc
        with:
          context: .
          push: true
          file: ./docker/Dockerfile.api
          tags: ${ steps.meta-api.outputs.tags }
          labels: ${ steps.meta-api.outputs.labels }

      - name: Extract metadata (tags, labels) for Docker
        id: meta-wrk
```

(continues on next page)

(continued from previous page)

```

uses: docker/metadata-action@98669ae865ea3cfffcbcaa878cf57c20bbf1c6c38
with:
  images: wjallen/mldata-wrk

- name: Build and push Docker image
  uses: docker/build-push-action@ad44023a93711e3deb337508980b4b5e9bcdc5dc
  with:
    context: .
    push: true
    file: ./docker/Dockerfile.wrk
    tags: ${{ steps.meta-wrk.outputs.tags }}
    labels: ${{ steps.meta-wrk.outputs.labels }}

```

This workflow waits is triggered when a new tag is pushed (tag: - '*'). As in the previous action, this one checks out the code and stages the sample data. Then, it uses the docker/login-action to log in to Docker Hub on the command line. The username and password can be set by navigating to Settings => Secrets => New Repository Secret within the project repository.

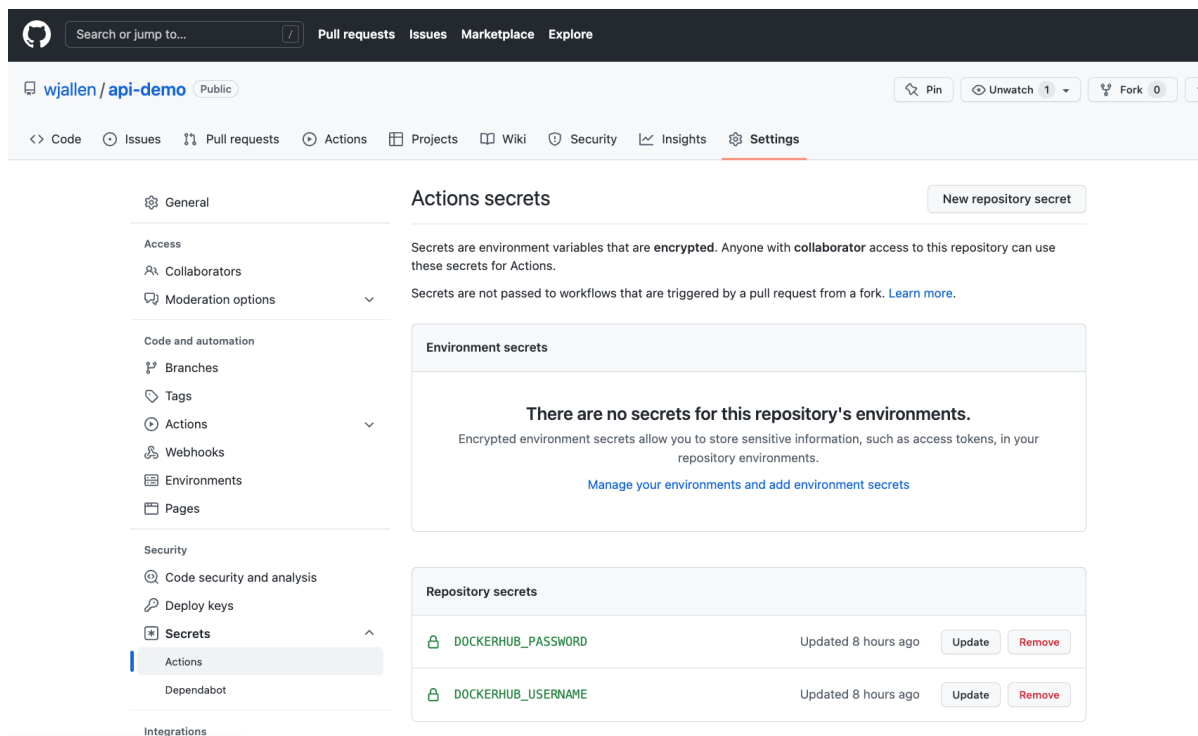


Fig. 3: Secrets are tied to specific repos.

Finally, this workflow extracts the tag from the environment and builds / pushes the API container, then builds / pushes the worker container both using actions from the GitHub Actions catalogue.

Tip: Don't re-invent the wheel when performing GitHub Actions. There is likely an existing action that already does what you're trying to do.

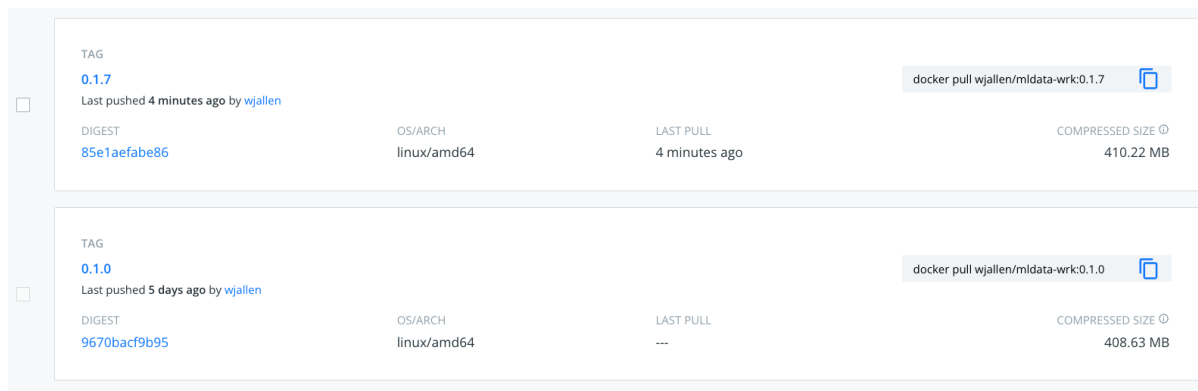
Trigger the Integration

To trigger the build in a real-world scenario, make some changes to your source code, push your modified code to GitHub and tag the release as X.Y.Z (whatever new tag is appropriate) to trigger another automated build:

```
[isp02]$ git add *
[isp02]$ git commit -m "added a new route to do something"
[isp02]$ git push
[isp02]$ git tag -a 0.1.1 -m "release version 0.1.1"
[isp02]$ git push origin 0.1.1
```

By default, the git push command does not transfer tags, so we are explicitly telling git to push the tag we created (0.1.1) to the remote (origin).

Now, check the online GitHub repo to make sure your change / tag is there, and check the Docker Hub repo to see if your new tag has been pushed.



<input type="checkbox"/>	<div>TAG</div> <div>0.1.7</div> <div>Last pushed 4 minutes ago by wjallen</div>	<div>OS/ARCH</div> <div>linux/amd64</div>	<div>LAST PULL</div> <div>4 minutes ago</div>	<div>COMPRESSED SIZE</div> <div>410.22 MB</div>
<input type="checkbox"/>	<div>TAG</div> <div>0.1.0</div> <div>Last pushed 5 days ago by wjallen</div>	<div>OS/ARCH</div> <div>linux/amd64</div>	<div>LAST PULL</div> <div>---</div>	<div>COMPRESSED SIZE</div> <div>408.63 MB</div>

Fig. 4: New tag automatically pushed.

9.4.6 Deploy to Kubernetes

The final step in our example is to update the image tag in our deployment YAML files in both test and prod, and apply them all. Apply to test (staging) first as one final check that things are working as expected. Then, deploy to prod. Because the old containers are Running right up until the moment the new containers are deployed, there is virtually no disruption in service.

Note: Some CI / CD services can even handle the deployment to Kubernetes following Docker image builds and passing tests.

9.4.7 Additional Resources

- [GitHub Actions Docs](#)
- [Demo Repository](#)

UNIT 10: SPECIAL TOPICS

Miscellaneous ideas and guides to support the concepts learned in this class and may help you in completing aspects of the final project.

10.1 Docker Compose, Revisited

Let's do a bit of rehash: docker-compose is a tool for defining and running multi-container Docker applications. With docker-compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration.

Here, we will revisit our containerized Flask and Redis services, fire them up with docker-compose, and then have our two services talk to each other.

When working with multiple containers, it can be difficult to manage the starting configuration along with the variables and links. Docker-compose is one orchestration tool for defining and running multi-container Docker applications.

10.1.1 Why Docker-Compose?

- Orchestration!
- Launch multiple containers with complex configurations at once
- Define the structure of your app, not the commands needed to run it!

10.1.2 Using Docker-Compose

Using docker-compose is a three-step process:

- Define images with Dockerfiles
- Define the services in a docker-compose.yml as containers with all of your options (e.g. image, port mapping, links, etc.)
- Run `docker-compose up` and it starts and runs your entire app

Three step process to use ... a bit more to actually build.

10.1.3 Orchestrating Redis

The first thing to do is create a new Docker build context for our app - this is the collection of files and folders that goes into the image(s) we build. Create a folder called `redis-docker` and directories called `config` and `data` within:

```
[isp02]$ mkdir redis-docker/
[isp02]$ mkdir redis-docker/config
[isp02]$ mkdir redis-docker/data
```

Next copy [this redis config file](#) into your config directory. We are going to put this into our Redis container, and it will allow us to customize the behavior of our database server, if desired.

```
[isp02]$ cd redis-docker
[isp02]$ wget -O config/redis.conf https://raw.githubusercontent.com/TACC/coe-332-sp21/
↪main/docs/week09/redis.conf
[isp02]$ ls config/
redis.conf
```

Now in your top directory, create a new file called `docker-compose.yml`. Populate the file with the following contents, being careful to preserve indentation, and replacing the username / port placeholders with your own:

```
---
version: '3'
services:
  redis:
    image: redis:latest
    container_name: <your username>-redis
    ports:
      - <your redis port>:6379
    volumes:
      - ./config/redis.conf:/redis.conf
    command: [ "redis-server", "/redis.conf" ]
```

10.1.4 Start the Redis Service

Bring up your Redis container with the following command:

```
[isp02]$ docker-compose -p <your username> up -d
```

Take note of the following options:

- `docker-compose up` looks for `docker-compose.yml` and starts the services described within
- `-p <your username>` gives the project a unique name, which will help avoid collisions with other student's containers
- `-d` puts it in daemon mode (runs in the background)

Check to see if your Redis database is up and the port is forwarding as you expect with the following:

```
[isp02]$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS
↪ NAMES
a1b2b6908a9   redis:5.0.0    "docker-entrypoint.s..."  58 seconds ago Up 55↪
↪seconds      0.0.0.0:6080->6379/tcp  charlie-redis
```

(continues on next page)


```

---
version: '3'
services:
  redis:
    image: redis:latest
    container_name: <your username>-redis
    ports:
      - <your redis port>:6379
    volumes:
      - ./config/redis.conf:/redis.conf
    command: [ "redis-server", "/redis.conf" ]
  web:
    build: .
    container_name: <your-username>-web
    ports:
      - <your flask port>:5000
    volumes:
      - ./data/data_file.json:/datafile.json

```

With these lines, you are adding a new service called 'web'. Take care to replace the placeholders with your assigned Redis port and Flask port numbers. Note Redis and Flask use default ports 6379 and 5000, respectively, inside the containers unless otherwise specified.

Also new to this service, we are using the build key to build a new Docker image based on the files / Dockerfile in this (.) directory. We need to pull in our web assets (wherever they are located - it may be different for each person) and Dockerfile from our previous exercises to this current directory.

```

[isp02]$ mkdir web
[isp02]$ cp ~/coe-332/web1/app.py ./web/
[isp02]$ cp ~/coe-332/web1/requirements.txt ./
[isp02]$ cp ~/coe-332/web1/data_file.json ./data/
[isp02]$ cp ~/coe-332/web1/Dockerfile ./

```

Now your directory structure should look like:

```

[isp02]$ tree .
.
├── config
│   └── redis.conf
├── data
│   └── data_file.json
├── docker-compose.yml
├── Dockerfile
├── web
│   ├── app.py
│   └── requirements.txt

```

This time when you start services, two containers will be created, one of which is built from the current directory.

```

[isp02]$ docker-compose -p charlie up -d
Creating network "charlie_default" with the default driver
Creating charlie-redis ... done
Creating charlie-web   ... done

```


10.1.6 Modify Python Redis Client

When you do `docker-compose up`, behind the scenes Docker creates a custom bridge network for each of your services to talk to one another. They can reach each other using the name of the service as the 'host', e.g.:

```
>>> rd = redis.StrictRedis(host='redis', port=6379, db=0)
```

Exercise

Connect your Flask container and your Redis container together using `docker-compose`, and curl the various endpoints to make sure it works.

Note: Be sure to change your Redis connection in your Flask App!

10.2 Dynamic Redis IPs

10.2.1 Updating the Flask API to use the Redis Service IP

In your flask code, you have a line that looks something like this:

```
rd=redis.StrictRedis(host='redis', port=6379, db=0)
```

Recall that the `host='<some_host>'` argument instructs the Redis client to use a particular network address (an IP address or a domain) to connect to Redis. We know from the lab that, in our k8s deployment, the Redis database will be available from the Redis service IP. We need to make sure that our flask API uses this API.

10.2.2 Option 1: Hard Code the Service IP Directly in the Python Code

This is the simplest approach. If our Redis service IP were `10.108.118.36` we would simply replace the above with:

```
rd=redis.StrictRedis(host='10.108.118.3', port=6379, db=0)
```

This works, but the problem is that we have to change the code every time the Redis service IP changes. It's true that we use services precisely because their IPs don't change, but as we move from our test to our prod environment (recall the discussion on environments from earlier), the Redis service IP will change. Once our code in the test environment has been tested, we want to be able to deploy it to prod exactly as is, without making any changes.

10.2.3 Option 2: Pass the IP as an Environment Variable

The better approach is to pass the Redis IP as an environment variable to our service. Environment variables are variables that get set in the shell and are available for programs. In python, the `os.environ` dictionary contains a key for every variable. So, we can use the following instead:

```
import os

redis_ip = os.environ.get('REDIS_IP')
if not redis_ip:
```

(continues on next page)

(continued from previous page)

```

raise Exception()
rd=redis.StrictRedis(host=redis_ip, port=6379, db=0)

```

This way, if we set an environment variable called REDIS_IP to our Redis service IP before starting our API, the flask code will automatically pick it up and use it.

In homework 5, you saw how to set environment variables in a k8s pod. We'll revisit this idea when discussing continuous integration.

10.3 Plotting with Matplotlib

10.3.1 What is Matplotlib

It's a graphing library for Python. It has a nice collection of tools that you can use to create anything from simple graphs, to scatter plots, to 3D graphs. It is used heavily in the scientific Python community for data visualization.

Let's plot a simple sin wave

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.linspace(0, 2*np.pi, 50)
5 plt.plot(x, np.sin(x))
6 plt.show() # we can't do this on our VM server
7 plt.savefig('my_sinwave.png')

```

we can keep plotting! Let's plot 2 graphs on the same axis

```

1 plt.plot(x, np.sin(x), np.sin(2*x))
2 plt.show()
3 plt.savefig('my_sinwavex2.png')

```

why stop now? Let's make the plot easier to read

```

1 plt.plot(x, np.sin(x), 'r-o', x, np.sin(2*x), 'g--')
2 plt.show()
3 plt.savefig('my_sinwavex2a.png')

```

other color combinations:

Colors:

- Blue – 'b'
- Green – 'g'
- Red – 'r'
- Cyan – 'c'
- Magenta – 'm'
- Yellow – 'y'
- Black – 'k' ('b' is taken by blue so the last letter is used)
- White – 'w'

Lines and markers:

- Lines:
 - Solid Line – ‘-’
 - Dashed – ‘--’
 - Dotted – ‘.’
 - Dash-dotted – ‘-.’
- Often Used Markers:
 - Point – ‘.’
 - Pixel – ‘,’
 - Circle – ‘o’
 - Square – ‘s’
 - Triangle – ‘^’

10.3.2 Subplots

using the subplot() function, we can plot two graphs at the same time within the same “canvas”. Think of the subplots as “tables”, each subplot is set with the number of rows, the number of columns, and the active area, the active areas are numbered left to right, then up to down.

```

1 plt.subplot(2, 1, 1) # (row, column, active area)
2 plt.plot(x, np.sin(x))
3 plt.subplot(2, 1, 2) # switch the active area
4 plt.plot(x, np.sin(2*x))
5 plt.show()
6 plt.savefig('my_sinwavex2b.png')
```

10.3.3 Scatter plots

```

1 y = np.sin(x)
2 plt.scatter(x,y)
3 plt.show()
4 plt.savefig('my_scattersin.png')
```

Let’s mix things up, using random numbers and add a colormap to a scatter plot

```

1 x = np.random.rand(1000)
2 y = np.random.rand(1000)
3 size = np.random.rand(1000) * 50
4 color = np.random.rand(1000)
5 plt.scatter(x, y, size, color)
6 plt.colorbar()
7 plt.show()
8 plt.savefig('my_scatterrandom.png')
```

We brought in two new parameters, size and color, which will vary the diameter and the color of our points. Then adding the colorbar() gives us a nice color legend to the side.

10.3.4 Histograms

A histogram is one of the simplest types of graphs to plot in Matplotlib. All you need to do is pass the `hist()` function an array of data. The second argument specifies the amount of bins to use. Bins are intervals of values that our data will fall into. The more bins, the more bars.

```
1 plt.hist(x, 50)
2 plt.show()
3 plt.savefig('my_histrandom.png')
```

10.3.5 Adding Labels and Legends

```
1 x = np.linspace(0, 2 * np.pi, 50)
2 plt.plot(x, np.sin(x), 'r-x', label='Sin(x)')
3 plt.plot(x, np.cos(x), 'g-^', label='Cos(x)')
4 plt.legend() # Display the legend.
5 plt.xlabel('Rads') # Add a label to the x-axis.
6 plt.ylabel('Amplitude') # Add a label to the y-axis.
7 plt.title('Sin and Cos Waves') # Add a graph title.
8 plt.show()
9 plt.savefig('my_labels_legends')
```

10.3.6 Redis and plots

you can “save” your plots to Redis, however the maximum size for a key/value is 512 mb and the sum of all your data (including files) must fit into main memory on the Redis server.

```
1 import redis
2 rd = redis.StrictRedis(host='172.17.0.1', port=6379, db=0)
3
4 # read the raw file bytes into a python object
5 file_bytes = open('/tmp/myfile.png', 'rb').read()
6
7 # set the file bytes as a key in Redis
8 rd.set('key', file_bytes)
```

10.4 Storing Images in Redis

As part of the final project, your worker may create an image of a plot. If it is created inside the Kubernetes worker pod, you’ll need a convenient way to retrieve that image back out of the worker container and into whatever container you curled from.

The easiest way to retrieve the image is for the worker to add the image back to the Redis db, and for the user to query the database with a flask route and retrieve the image. This would be the general workflow:

1. The user submits a curl request from, e.g., the py-debug pod to the flask api
2. The flask api creates a new job entry in the redis db, and adds the UUID to the queue
3. The worker picks up the job, and creates a plot
4. The worker saves the plot in the redis db under the job entry

- The user curls a new route to download the image from the db

10.4.1 Initiate a Job

Imagine that when a user submits a job, an entry is created in the jobs db of the following form:

```
[isp02]$ curl localhost:5000/submit -X POST -H "Content-Type: application/json" -d '{
  ↪ "start": "2001", "end": "2021"}'
Job 161207aa-9fe7-4caa-95b8-27f5bcbb16e7 successfully submitted
```

```
[isp02]$ curl localhost:5000/jobs
{
  "161207aa-9fe7-4caa-95b8-27f5bcbb16e7": {
    "status": "submitted",
    "start": "2001",
    "end": "2021"
  }
}
```

10.4.2 Add an Image to a Redis DB

The worker picks up the job, performs the analysis based on the 'start' and 'end' dates, and generates a plot. An example of using matplotlib to write and save a plot to file might look like:

```
1 import matplotlib.pyplot as plt
2
3 x_values_to_plot = []
4 y_values_to_plot = []
5
6 for key in raw_data.keys():      # raw_data.keys() is a client to the raw data stored
  ↪ in redis
7     if (int(start) <= key['date'] <= int(end)):
8         x_values_to_plot.append(key['interesting_property_1'])
9         y_values_to_plot.append(key['interesting_property_2'])
10
11 plt.scatter(x_values_to_plot, y_values_to_plot)
12 plt.savefig('/output_image.png')
```

Warning: The code above should be considered pseudo code and not copy/pasted directly. Depending on how your databases are set up, client names will probably be different, you may need to decode values, and you may need to cast type on values.

Now that an image has been generated, consider the following code that will open up the image and add it to the Redis db:

```
1 with open('/output_image.png', 'rb') as f:
2     img = f.read()
3
4 rd.hset(jobid, 'image', img)
5 rd.hset(jobid, 'status', 'finished')
```

Note: If anyone has a way to get the png file out of the Matplotlib object without saving to file, please share!

10.4.3 Retrieve the Image with a Flask Route

Now that the image has been added back to the jobs database, you can expect this type of data structure to exist:

```
{
  "161207aa-9fe7-4caa-95b8-27f5bcbb16e7": {
    "status": "finished",
    "start": "2001",
    "end": "2021",
    "image": <binary image data>
  }
}
```

It would not be a good idea to show that binary image data with the rest of the text output when querying a /jobs route - it would look like a bunch of random characters. Rather, write a new route to download just the image given the job ID:

```
1 from flask import Flask, request, send_file
2
3 @app.route('/download/<jobid>', methods=['GET'])
4 def download(jobid):
5     path = f'/app/{jobid}.png'
6     with open(path, 'wb') as f:
7         f.write(rd.hget(jobid, 'image'))
8     return send_file(path, mimetype='image/png', as_attachment=True)
```

Flask has a method called 'send_file' which can return a local file, in this case meaning a file that is saved inside the Flask container. So first, open a file handle to save the image file inside the Flask container, then return the image as mimetype='image/png'.

The setup above will print the binary code to the console, so the user should redirect the output to file like:

```
[isp02]$ curl localhost:5000/download/161207aa-9fe7-4caa-95b8-27f5bcbb16e7 > output.png
[isp02]$ ls
output.png
```

Note: If anyone has a way to download the image to file automatically without redirecting to file, please share!

HOMEWORK 01

Due Date: Tuesday, Feb 1, by 11:00am CST

11.1 As Good as Git Gets

Your task for the first homework is to create a new Git repository and populate it with several files / folders. This Git repository must be pushed to GitHub, and it will be used for the rest of the semester to turn in your homeworks.

Here are the requirements for the Git repository:

- Must contain a top level `README.md` with a simple description of what the repository contains
- Must contain a directory called `homework01`
- Must contain a minimum of two different Python scripts, each solving one Exercise at the end of the [Python Refresher guide](#)
- The directory `homework01` must also contain a `README.md` file with a description of the Python scripts (contents may vary between students)
- **Must contain a minimum of two commits - use git log to see commits**
- Must be pushed to GitHub, and be either public or private (if private, make sure to add wallen@tacc.utexas.edu as an collaborator on the repo)

A sample Git repository may contain the following files after completing homework 01:

```
my-coe332-hws/
├── homework01
│   ├── list_generator.py
│   ├── README.md          # specifically describes the contents of the homework01
└── directory
    ├── word_reader.py
    └── README.md          # generally describes the whole repo
```

11.2 What to Turn In

Send an email to wallen@tacc.utexas.edu with the link to your GitHub repository and include “Homework 01” in the subject line. We will clone all of your repos at the due date / time for evaluation.

11.3 Additional Resources

- [Python Refresher guide](#)
- Please find us in the class Slack channel if you have any questions!

HOMEWORK 02

Due Date: Thursday, Feb 10, by 11:00am CST

12.1 Return of the JSON

Scenario: You are operating a robotic vehicle on Mars and the task for today is to investigate **five** meteorite landing sites in [Syrtis Major](#).

12.1.1 PART 1

In the first part of this homework, you will write a Python script to randomly generate latitude, longitude, and compositions for the five meteorite landing sites. The requirements are as follows:

- Generate five random pairs of latitudes (between 16.0 - 18.0 degrees North) and longitudes (between 82.0 - 84.0 degrees East) in decimal notation
- For each landing site, also randomly choose a meteorite composition from the list ["stony", "iron", "stony-iron"]
- Assemble these data into a *dictionary with one key, "sites", whose value is a list of dictionaries*
- Use the Python json library to save the data to a JSON file. For example, your data structure may look like:

```
{
  "sites": [
    {
      "site_id": 1,
      "latitude": 17.93705170143149,
      "longitude": 83.36448444826725,
      "composition": "stony"
    },
    {
      "site_id": 2,
      "latitude": 16.714833623042153,
      "longitude": 82.84554246756586,
      "composition": "iron"
    },
    ... etc
  ]
}
```

12.1.2 PART 2

In the second part of this homework, you will write a **new** Python script that reads in the meteorite site JSON file and calculates the time required to visit and take samples from the five sites in order. The requirements are as follows:

- The second Python script should use the `json` library to read in the data generated in part 1 and store it as a dictionary
- Assume the robot starts at latitude / longitude {16.0, 82.0}
- Assume the robot visits the five sites in the order of the list index
- Assume the max robot speed is 10 km per hour
- Assume that Mars is a sphere (radius = 3389.5 km) and use the great-circle distance algorithm to calculate distance between points (check Slack for a hint on this)
- When the robot stops to take a sample of each meteorite, the amount of time it stops depends on the composition of the meteorite. Stony meteorites take 1 hour to sample, iron meteorites take 2 hours to sample, and stony-iron meteorites take 3 hours to sample
- The trip is “over” after sampling the last meteorite. Print some descriptive info for each leg of the trip, plus summary info for the whole trip. Sample output might look similar to:

```
leg = 1, time to travel = 11.75 hr, time to sample = 1 hr
leg = 2, time to travel = 3.43 hr, time to sample = 2 hr
leg = 3, time to travel = 4.53 hr, time to sample = 1 hr
leg = 4, time to travel = 6.04 hr, time to sample = 2 hr
leg = 5, time to travel = 10.43 hr, time to sample = 3 hr
=====
number of legs = 5, total time elapsed = 45.17 hr
```

12.1.3 PART 3

Your homework 02 files must be within a new subdirectory called `homework02` in your COE332 homeworks repository on GitHub. The directory should contain the two Python scripts and a `README.md` file. The README should be descriptive, use proper grammar, and contain enough instructions so anyone else could clone the repository and figure out what the scripts do and how to run them. General guidelines to follow for the README for homework 02 are:

- Descriptive title
- High-level description of the folder contents / project objective. I.e. why does this exist and why is it important? (2-3 sentences)
- Specific description of the individual python scripts (1-2 sentences each)
- Instructions to run the code from start to finish, plus how to interpret the results (2-3 sentences)
- Try to use markdown styles to your advantage, give the sections headers, use code blocks where appropriate, etc.

Remember, the README is your chance to document for yourself and explain to others why the project is important, what the code is, and how to use it / interpret the outputs / etc. This is a *software engineering and design* class, so we are not just checking to see if your code works. We are also evaluating the design of the overall submission, including how well the project is described in the README.

12.2 What to Turn In

A sample Git repository may contain the following new files after completing homework 02:

```
my-coe332-hws/  
├── homework01  
│   ├── list_generator.py  
│   ├── README.md  
│   └── word_reader.py  
├── homework02  
│   ├── calculate_trip.py    # your file names may vary  
│   ├── generate_sites.py  
│   └── README.md  
└── README.md
```

There is no need to email the link to your homework repo again, as we should have it on file from the first homework. We will re-clone the same repo as before at the due date / time for evaluation.

12.3 Additional Resources

- [JSON guide](#)
- [Latitude and Longitude as decimals](#)
- [Great-circle distance formula](#)
- [Markdown syntax](#)
- [Tips on writing a good README](#)
- Please find us in the class Slack channel if you have any questions!

HOMEWORK 03

Due Date: Thursday, Feb 17, by 11:00am CST

13.1 2001: A Space Turbidity

Scenario: Your robot has finished collecting its five meteorite samples and has taken them back to the Mars lab for analysis. In order to analyze the samples, however, you need clean water. You must check the latest water quality data to assess whether it is safe to analyze samples, or if the Mars lab should go on a boil water notice.

13.1.1 PART 1

For the first part of this homework, download the water quality data set from [this link](#). The data is a JSON dictionary with one key, 'turbidity_data', whose value is a time series list of dictionaries. Each dictionary in the list has the same set of keys. A sample of the data looks like:

```
{
  "turbidity_data": [
    {
      "datetime": "2022-02-01 00:00",
      "sample_volume": 1.19,
      "calibration_constant": 1.022,
      "detector_current": 1.137,
      "analyzed_by": "C. Milligan"
    },
    {
      "datetime": "2022-02-01 01:00",
      "sample_volume": 1.15,
      "calibration_constant": 0.975,
      "detector_current": 1.141,
      "analyzed_by": "C. Milligan"
    },
    ... etc
  ]
}
```

- **DO NOT** commit this data set to your homework repository
- **DO** provide instructions in your README on how to download this data set

Note: Turbidity is caused by particles suspended or dissolved in water that scatter light making the water appear cloudy or murky. [Source](#) ¹

13.1.2 PART 2

Write a Python3 script that reads in the water quality data set, and prints three key pieces of information to screen: (1) the current water turbidity (taken as the average of the most recent five data points), (2) whether that turbidity is below a safe threshold, and (3) the minimum time required for turbidity to fall below the safe threshold (if it is already below the safe threshold, the script would report 0 hours). Here are the requirements for this Python3 script:

- The script must have an appropriate `main()` function that is only called when executing your script directly
- The script must have a minimum of two additional functions:
 - A function to calculate turbidity using equation 1 below
 - A function to calculate minimum time to fall below threshold turbidity using equation 2 below
- The turbidity threshold for safe water is a constant, 1.0 NTU
- The decay factor per hour is a constant, 2% or 0.02 expressed as a decimal
- Each function (except `main()`) must have a concise docstring containing a description of the function, arguments, and return values following the format in the [Google Style Guide](#)
- Each function definition (except `main()`) must contain type hints
- The second line of the output (see examples below) must come from a log message using the logging module

Equation 1: The equation we will use for turbidity is based on readings taken by a nephelometer ([Source2](#)):

```
T = a0 * I90
T = Turbidity in NTU Units (0 - 40)
a0 = Calibration constant
I90 = Ninety degree detector current
```

Equation 2: The equation we will use for minimum time to return below a safe threshold is expressed as an inequality, and it is a standard exponential decay function:

```
Ts > T0(1-d)**b
Ts = Turbidity threshold for safe water
T0 = Current turbidity
d = decay factor per hour, expressed as a decimal
b = hours elapsed
```

When executing this script, the output might look similar to one of the following two code blocks, depending on whether turbidity is above or below the safe threshold:

```
Average turbidity based on most recent five measurements = 1.1992 NTU
Warning: Turbidity is above threshold for safe use
Minimum time required to return below a safe threshold = 8.99 hours
```

```
Average turbidity based on most recent five measurements = 0.9852 NTU
Info: Turbidity is below threshold for safe use
Minimum time required to return below a safe threshold = 0 hours
```

13.1.3 PART 3

In a new Python3 script, write unit tests to test your functions described above. The test script must be appropriately named and prefixed with `test_`. The test script must also work with `pytest`. There must be a minimum of **five** tests associated with each function, some of which should perform simple sanity checks that the math is correct, and others should perform more complicated checking including that types returned and exceptions thrown match what are expected.

13.1.4 PART 4

The homework must also include a README file. The README should be descriptive, use proper grammar, and contain enough instructions so anyone else could clone the repository and figure out what the script does and how to run it. General guidelines to follow for the README are:

- Descriptive title
- High-level description of the folder contents / project objective. I.e. why does this exist and why is it important? (2-3 sentences)
- Instructions on how to download the data set from the original source
- Specific description of the python script (1-2 sentences)
- Instructions to run the code from start to finish, plus how to interpret the results (2-3 sentences) (Example output would help a lot to explain how to interpret the results)
- Try to use markdown styles to your advantage, give the sections headers, use code blocks where appropriate, etc.

Remember, the README is your chance to document for yourself and explain to others why the project is important, what the code is, and how to use it / interpret the outputs / etc. This is a *software engineering and design* class, so we are not just checking to see if your code works. We are also evaluating the design of the overall submission, including how well the project is described in the README.

13.2 What to Turn In

A sample Git repository may contain the following new files after completing homework 03:

```
my-coe332-hws/
├── homework01
│   └── ...
├── homework02
│   └── ...
├── homework03
│   ├── analyze_water.py      # your file names may vary
│   ├── README.md
│   └── test_analyze_water.py
└── README.md
```

There is no need to email the link to your homework repo again, as we should have it on file from the first homework. We will re-clone the same repo as before at the due date / time for evaluation.

13.3 Additional Resources

- [Water quality data](#)
- [Water turbidity equations](#)
- [Google style guide](#)

HOMEWORK 04

Due Date: Tuesday, Mar 1, by 11:00am CST

14.1 Once Upon a Time in Containers

Scenario: You've been working on some code to analyze Meteorite Landing data. You wrote a few functions, you wrote some unit tests, you are satisfied with how everything is working together, and now you are ready to release your application to the world.

14.1.1 PART 1

Gather a recent copy of the `ml_data_analysis.py` script and the `Meteorite_Landings.json` data. You can start with the copy linked in the beginning of the [Advanced Containers guide](#), or you can use your own. Either way, for this homework you will need to make a couple small modifications to the Python3 script:

- Modify the script to take the name of the JSON file as a command line argument
- Have the script print out summary hemisphere data rather than individual hemisphere data for each meteor (see sample below). You will likely want to keep the functionality of the `check_hemisphere()` function the same, and do the actual counting of each quadrant for summary in the `main()` function.
- Print out some other text and formatting to make the output look a little nicer to read (see sample below)
- Update the tests in your `pytest` suite (if necessary) to make sure they are still passing with all the code changes. You should have a minimum of five tests for each of the three functions in `ml_data_analysis.py`

After the updates above, running your script against a representative data set might give output similar to the following (it does not have to match this exactly, it should just be informative and easy to read):

```
[isp02]$ ./ml_data_analysis Meteorite_Landings.json
Summary data following meteorite analysis:

Average mass of 30 meteor(s):
83857.3 grams

Hemisphere summary data:
There were 21 meteors found in the Northern & Eastern quadrant
There were 6 meteors found in the Northern & Western quadrant
There were 0 meteors found in the Southern & Eastern quadrant
There were 3 meteors found in the Southern & Western quadrant
```

(continues on next page)

(continued from previous page)

```
Class summary data:
The L5 class was found 1 times
The H6 class was found 1 times
The EH4 class was found 2 times
The Acapulcoite class was found 1 times
The L6 class was found 6 times
... etc
```

14.1.2 PART 2

Write a Dockerfile to containerize your `ml_data_analysis.py` script, the test script (for pytest), and a copy of the example data set. Build an image with the version tag `hw04` and push it to your namespace in Docker Hub.

Important: For this homework, use the version tag `hw04` exactly as it appears here. We will specifically be looking for that tag. No, do not use `'hw4'` or `'hw-04'` or any other similar spelling.

14.1.3 PART 3

You've written the code, you've containerized it, now you need to tell other people how to use it. The problem is that you are not quite sure *how* people will want to use this code. In the real world, developers and end users will find your repo, and some of them will want to use your pre-built Docker image to run the code, while others will want to start from your Dockerfile and build their own image. The challenge is to write a README that caters to all audiences.

Write a README with the standard sections from previous homeworks: there should be a descriptive title, there should be a high level description of the project, there should be concise descriptions of the main files within, and you should be using Markdown styles and formatting to your advantage.

The section of the README with run instructions will be a little more challenging. There should be instructions to, at a minimum:

- Pull and use your existing image on Docker Hub
- Build an image from your Dockerfile
- Run the containerized code against the sample data inside the container
- Run the containerized code against user-provided data that they may have found on the web
- Run the containerized test suite with pytest

Give example commands and expected outputs in code blocks.

Finally, your README should also have a section to describe the expected input data. You need to provide enough information to the reader so they know what their input data should look like. Showing a snippet of the data in a code block would be a really good idea here. You may also want to mention that there is some additional Meteorite Landing data available [at this link](#). **DO NOT** embed this data in your container. **DO** provide instructions to users to download this data and run the containerized scripts against this data.

14.2 What to Turn In

A sample Git repository may contain the following new files after completing homework 04:

```
my-coe332-hws/  
├── homework01  
│   └── ...  
├── homework02  
│   └── ...  
├── homework03  
│   └── ...  
├── homework04  
│   ├── Dockerfile           # your file names may vary  
│   ├── ml_data_analysis.py  
│   ├── README.md  
│   ├── Meteorite_Landings.json  
│   └── test_ml_data_analysis.py  
└── README.md
```

14.3 Additional Resources

- [Extra Meteorite Landing Data](#)

MIDTERM PROJECT

Due Date: Tuesday, March 22, by 11:00am CDT

15.1 O ISS, Where Art Thou?

Scenario: You have found an abundance of positional data for the International Space Station (ISS). It is full of interesting information including ISS position and velocity data at given times, as well as when the ISS can be seen over select cities. It is a challenge, however, to sift through the data manually to find what you are looking for. Your objective is to build a containerized Flask application for querying and returning interesting information from the ISS data set.

15.1.1 PART 1

Write and containerize a Flask application for tracking ISS position and sightings. The application should load in two data sets (referred to below as the “positional data” and the “sighting data”).

- The ISS positional data set [found here](#), under the link ‘Public Distribution File’
- One of the 18 sighting data sets, [also found here](#) under various names. Find the name of the sighting data set that was assigned to you in the spreadsheet uploaded to Canvas

The Flask application should have routes to return:

- Information on how to interact with the application
- All Epochs in the positional data
- All information about a specific Epoch in the positional data
- All Countries from the sighting data
- All information about a specific Country in the sighting data
- All Regions associated with a given Country in the sighting data
- All information about a specific Region in the sighting data
- All Cities associated with a given Country and Region in the sighting data
- All information about a specific City in the sighting data

In addition, a special route is needed to load the data from file into memory. This must be a **POST** endpoint. Refer to the unit on [XML](#) for tips on reading XML data into a dictionary.

All of the normal Python3 script best practices apply:

- Write appropriately formatted doc strings

- Use type annotations where appropriate
- Use log messages to report when routes are queried

15.1.2 PART 2

Dockerfile. In addition to the application Python script, you must write and include a Dockerfile. The Dockerfile should containerize the application and both data sets (positional data and sighting data), and it should set a default command to launch the Flask application.

Pytest. A unit test file must exist, it must be compatible with Pytest, and it must also be included in the container. Each function (except `main()`) in the Flask application must be adequately tested.

Makefile. A Makefile must be written with targets to build a container and to start the containerized Flask application (at a minimum). Please make sure to configure the Makefile to use the Flask port you were assigned (in the range 5001-5040).

15.1.3 PART 3

This Midterm has two required written components:

README. Standard README guidelines apply. The README should have a descriptive title (not “Midterm Project”), a short summary / description of the project, short descriptions of any very important files that the reader should know about, instructions to download the data and build the containerized app, instructions to pull a pre-containerized copy of the app from Docker Hub, instructions to interact with the application, and guidelines on how to interpret the results. Be succinct, use Markdown styles, and show example commands / expected output.

Write Up. You must also turn in a written document (as a PDF) describing the project. Where as the README is typically more succinct and targeted towards developers / users of the application, the written document should be more verbose and targeted towards a non-user, but technically savvy layperson. For example, in the README you might instruct a user: “Run the application by performing xyz”. In the Write Up, you would instead describe what users do: “Users of the application can run it by performing xyz”. In other words, write this document as if you are describing to your fellow engineering students what you did in this class for your Midterm project.

In the Write Up, include some narrative about the motivation of the project and why it is an interesting or important application to have. You must also include a short section on “Ethical and Professional Responsibilities in Engineering Situations”. As a suggestion, you may consider describing the importance of citing data or the importance of verifying the quality and accuracy of data that goes into applications such as these. We strongly encourage you to come up with other Ethical and Professional Responsibilities that might apply here.

Please make sure to appropriately cite the [data source](#) in both the README and the written document.

15.2 What to Turn In

This Midterm project should be pushed into a standalone repo with a descriptive name (not “coe332-midterm”). It should not be put into your existing homework repo. A sample Git repository may contain the following after completing the Midterm:

```
descriptive-repo-name/  
├── Dockerfile  
├── Makefile  
├── README.md  
├── app.py  
└── pytest_app.py
```

Do not include the raw XML data as part of your repo.

Send an email to wallen@tacc.utexas.edu with the written PDF summary of the project attached plus a link to your new GitHub repository. Please include “Midterm Project” in the subject line. We will clone all of your repos at the due date / time for evaluation.

15.3 Additional Resources

- [NASA Data Set](#)
- [Unit on XML](#)
- Please find us in the class Slack channel if you have any questions!

HOMEWORK 05

Due Date: Tuesday, Apr 5, by 11:00am CDT

16.1 Back to the Flask

A database server (like Redis) is a critical addition to our applications for data persistence. In this short homework assignment, we will launch a Redis container, then build a small Flask app to load data into and retrieve data from the database.

16.1.1 PART 1

Launch a new containerized instance of a Redis database server using the stock `redis:6` image. Be sure to:

- Run the container in the background
- Connect your assigned Redis port on ISP to the default Redis port inside the container
- Mount a local folder to the `/data` folder inside the container for Redis to write its backups to
- Configure the database server (as we did in class) to save to 1 backup file every 1 second

The command to do the above plus a description of what it is doing should be included in the README.

Tip: After creating the Redis container, do:

- `docker inspect <container ID> | grep IPAddress` to find the IP (e.g. `172.xx.x.x`)
 - Create the redis client in your Flask app as: `redis.Redis(host='172.xx.x.x', port=6379)`
 - Replace `172.xx.x.x` with the actual IP you find in the first part
 - Use the default port 6379 when communicating over the docker bridge network
 - [More info here](#)
-

16.1.2 PART 2

Write a containerized Flask app with one route that handles both a POST and a GET request.

- The route should be called `/data`
- A POST request to `/data` should load some Meteorite Landings data into your Redis database instance
- A GET request to `/data` should read the data out of Redis and return it as a JSON list
- For bonus points, implement an optional `start` query parameter that takes an integer and returns the Meteorite Landing data starting at that index. Make sure to handle the case where `start` is provided but is not an integer!

Use the Meteorite Landing data found at this link:

https://raw.githubusercontent.com/wjallen/coe332-sample-data/main/ML_Data_Sample.json

Please include docstrings and type annotations in your Python3 code.

16.1.3 PART 3

A Dockerfile and a README should be included in your homework repository. The usual Dockerfile and README guidelines apply. Make sure to provide instructions to launch the Redis database, instructions to pull/build/launch the Flask app, instructions to perform POST and GET requests to your Flask app, and a description of the data.

16.2 What to Turn In

A sample Git repository may contain the following new files after completing homework 05:

```
my-coe332-hws/
├── homework01
│   └── ...
├── homework02
│   └── ...
├── ... etc
├── homework05
│   ├── Dockerfile           # your file names may vary
│   ├── app.py
│   └── README.md
└── README.md
```

16.3 Additional Resources

- [Meteorite Landing Data](#)
- [Persistence in Redis](#)
- [Docker Bridge Network](#)

HOMEWORK 06

Due Date: Tuesday, Apr 19, by 11:00am CDT

17.1 Into the Kubernetes-Verse

The objective of this homework is to complete the in-class exercise from the end of the [Kubernetes section](#). You must deploy the Flask app and Redis server from [Homework 05](#) in Kubernetes.

17.1.1 PART 1

Complete the in-class exercise from the end of the [Kubernetes section](#). Successful completion of this exercise requires:

- A persistent volume claim for your Redis data
- A deployment for your Redis database
- A service for your Redis database
- A deployment for your Flask API
- A service for your Flask API

17.1.2 PART 2

The Dockerfile and app.py from Homework 05 should be copied into this new directory, and updated if necessary. A new README should be included in this homework repository. Standard README directions apply. No need to include instruction on building or pushing the docker image to Docker Hub. This time, focus on giving instructions for deploying the software system to a Kubernetes cluster using the containerized Flask app that you have previously built and pushed to Docker Hub in your own namespace.

17.2 What to Turn In

A sample Git repository may contain the following new files after completing homework 06:

```
my-coe332-hws/  
├── homework01  
│   └── ...  
├── homework02  
│   └── ...  
└── ...
```

(continues on next page)

(continued from previous page)

```
├── ... etc
├── homework06
│   ├── Dockerfile          # your file names may vary
│   ├── app.py
│   ├── README.md
│   └── *.yaml              # five YAML files for Part 1
└── README.md
```

17.3 Additional Resources

- [Homework 06 Lab / Steps.](#)

HOMEWORK 07

Due Date: Thursday, Apr 28, by 11:00am CDT

18.1 Million Dollar Diagram

This homework is designed to get you a jumpstart on the software diagram for your final project. There are no coding elements required. The only two things to turn in in are a software diagram (image file) and a README file.

18.1.1 PART 1

Create a software diagram showing off some interesting aspect of your Midterm project. You may choose which part(s) to illustrate and what type of software diagram you wish to create. See the class materials for [examples](#).

The diagram should be saved as an image file (e.g. png or something that will render in the README markdown file).

18.1.2 PART 2

The README should have a detailed description of the design diagram and a link to your midterm project that it is referencing. The target audience for the description is the same as the target audience for your final project write ups - your engineering peers who are not taking this class.

18.2 What to Turn In

A sample Git repository may contain the following new files after completing homework 07:

```
my-coe332-hws/
├── homework01
│   └── ...
├── homework02
│   └── ...
├── ... etc
├── homework07
│   ├── image.png          # your file names may vary
│   └── README.md
└── README.md
```

18.3 Additional Resources

- [Software Design Diagram Materials](#)

HOMEWORK 08

Due Date: Thursday, May 5, by 11:00am CDT

19.1 The GitHub Redemption

This final homework involves making corrections on your past seven homeworks. Over the course of the semester, we have emphasized organization, using descriptive titles in your READMEs, and not including large raw data files in the repos, among other things. This homework is your opportunity to go back through your first seven homeworks and give them one final polish.

19.1.1 EXPECTATIONS

- Your homework repo should contain one very brief top-level README which mentions that this is a homework repo for this class.
- Your top level repo should have exactly seven folders named “homework01” through “homework07” and nothing else. Naming must be consistent - i.e. don’t use a mix of “hw01” and “homework02” and “homework3”.
- Each of the seven homework folders should have a README with a descriptive title, not “Homework N”. E.g. you might rename “Homework 01” to “Python Test Scripts” or something similar.
- Each folder should include files necessary for the homework, and should not include superfluous or temporary files (e.g. anything ending in .swp, ~, __pycache__ folders, etc)
- No need to edit any old Python scripts, Dockerfiles, YAML files - they will not be re-evaluated. The only files you should be editing for this are the READMEs. Everything else is just removing or renaming files and / or folders.

19.2 What to Turn In

Send an e-mail to wallen@tacc.utexas.edu when you’ve completed this homework. Please include a link to your GitHub repo in the body of the e-mail, and please include “Homework 08” in the subject line.

FINAL PROJECT

Due Date for Part 1: Tuesday, April 12, by 11:00am CDT

Due Date for Parts 2-4: Thursday, May 12, by 9:00am CDT

20.1 REST APIs: Endgame

You've previously created a REST API interface to ISS positional data. Expanding on that idea, the final project will build an interface to a new data set of your choosing, and it will add a few more interesting features. The final application will be deployed to the cloud (Kubernetes) and made publicly accessible.

Warning: As long as this warning message is here, the content below is subject to change. We will notify the class any time a substantial change is made.

20.1.1 PART 1: Pitch

Form groups of **two or three** students to work collaboratively on the final project. You may choose your own groups, or you may ask the instructors to assign you to a group. Please let us know what groups you are in, or if you would like to be assigned to a group ASAP.

The first part of the final project is to identify an interesting data set that you want to work on. The data set should be Engineering focused, broadly defined. The data set should also be amenable to CRUD operations and some sort of analysis (see below). There are several links at the bottom of this page to Engineering-focused data sets, but you may look elsewhere too. Once your group has identified a potential data set to work on, write up a ~1 page summary of the proposed title of your project, list of group members, and a description of the data. Then schedule a ~10 minute meeting with at least one of the instructors in order to “pitch” your project. We want to know what the source is of the data, see what the data looks like, and hear what is your proposal for working on the data.

20.1.2 PART 2: Code Repository

The final project will involve building a REST API front-end to a (preferably) time series data set that allows for basic CRUD - Create, Read, Update, Delete - operations and also allows users to submit analysis jobs. At a minimum, the application should support on the back-end an analysis job to create a plot of the data. It must be hosted on the Kubernetes cluster. Extra credit may be given if the application supports additional types of analysis jobs, deploys over multiple environments (test and prod), or has an innovative user interface. The application will consist of:

- A front-end REST API with endpoints for each CRUD operation and for submitting a job to plot data and retrieving the resulting plot

- At least two back-end workers to perform the analysis / plotting job(s)
- A Redis database and queue for linking front-end and back-end processes

The project must also include a well-written README following all the guidelines given in previous class assignments. This README should emphasize two sections: instructions for deploying and testing the applicaiton, and instructions for using the application.

Other files including Kubernetes configuration files, Dockerfile(s), Makefile(s), and functional test file(s) will be expected (see rubric).

20.1.3 PART 3: Write Up

We are looking for a 10-11 page written document (as a PDF) describing the project. The written document should be verbose and targeted towards a non-user, but technically savvy layperson (e.g. one of your fellow engineering students who is not taking this class). Be sure to include narrative about the motivation of the project, why it is interesting or important to have the application you developed, a detailed description of the data (include citation), and you must include a short section on “Ethical and Professional Responsibilities in Engineering Situations”.

NEW REQUIREMENT: The write up must contain at least one design diagram showing the different parts of your software system and how they are connected. There should be supporting text to describe the diagram. We will cover some materials on creating design diagrams in future lectures.

20.1.4 PART 4: Video Demo

Prepare a < 10 minute video demo of the application. Use zoom to screen share and record your narration of the process. At a minimum, we want to see you deploy the application to Kubernetes, curl various routes to demonstrate CRUD operations, curl the appropriate routes to submit a an analysis job and retrieve and display the results, and highlight anything else you think is interesting or unique about your application.

20.2 What to Turn In

This Final project should be pushed into a standalone repo with a descriptive name. It should not be part of your existing homework repo. A sample Git repository may contain the following after completing the Final:

```
repo-name/
├── docker
│   ├── docker-compose.yml
│   ├── Dockerfile.api
│   └── Dockerfile.wrk
├── kubernetes
│   └── prod
│       ├── app-prod-api-deployment.yml
│       ├── app-prod-api-service.yml
│       ├── app-prod-db-deployment.yml
│       ├── app-prod-db-pvc.yml
│       ├── app-prod-db-service.yml
│       └── app-prod-wrk-deployment.yml
├── Makefile
├── README.md
├── requirements.txt
└── src
```

(continues on next page)

(continued from previous page)

```
├── flask_api.py
├── jobs.py      # you may or may not have this file, see note below
├── worker.py
├── test
└── test_flask.py
```

Send an email to wallen@tacc.utexas.edu with the written PDF summary of the project attached plus a link to your new GitHub repository plus a link to download the zoom recording. Please include “Final Project” in the subject line. We will clone all of your repos at the due date / time for evaluation.

Note: Note from Slack: There has been some confusion about whether or not to include jobs.py in the final project, because I did not explicitly list that file under ‘what to turn in’ in the final project description. If you have it and it is part of your software system, then yes absolutely include it in the repo. We are usually careful to say “A sample git repo may contain...”, meaning the example we give is an example only, not set in stone. Some people may arrange their python functions into a jobs.py file, and some people may not. It is dependent on your situation and how you choose to design your system. But yes, at the end of the day please commit and push every file that is an essential part of the whole system

20.3 Additional Resources

- [NASA Machine Learning Data Sets](#)
- [NASA Earth Data Sets](#)
- [Other NASA Data Sets](#)
- [Kaggle](#)
- [US Govt Data](#)
- [Data World Engineering Data Sets](#)
- [List of a bunch of different data sources](#)
- Please find us in the class Slack channel if you have any questions!

ADDITIONAL RESOURCES

- Slack: <https://tacc-learn.slack.com/>
- Class Repo: <https://coe-332-sp22.readthedocs.io/>
- Canvas: <https://utexas.instructure.com/courses/1326929>