Chunkify (< 1 hour):
Resources used: None

```
def chunkify(string, chunck_length, fill):
    # improper parameter check
    if string is None or string == "":
        pass
    elif chunck_length < 1:
        pass
    elif fill is None or fill == "":
        fill = " "
    elif len(fill) > 1:
        fill = fill[0]
    chunck_length = int(chunck_length)

    chunks = []
    numFill = (chunck_length - (len(string) % chunck_length)) % chunck_length
    for i in range(0, numFill):
        string += fill
    for i in range(0, int(len(string) / chunck_length)):
        chunk = string[(chunck_length * i) : (chunck_length * (i + 1))]
        chunks.append(chunk)
    print(chunks)
    pass
```

Coding to an API (~5-6 hours):
Resources used:

1. How to create Trello Cards from the Command Line (with a ~10 minute set-up) (github.com) - for the idea how to set it up.
2. The Trello REST API (atlassian.com) - for how to implement it

File: archive.tar

Code review (~4 hours):
Resources used: None

File: review.txt

Python Failure (~ 1 hour):
Resources used: None

If a piece of software written in python crashed with a stack trace, I would first analyze the stack trace to identify the point in the process where the crash occurred. Next I would look at the line that threw the error, both in the stack trace and in the code (typically the last entry in the stack trace). A brief description of the error should be provided with the stack trace, so I would try to deduct why that line threw that error. If the error isn't evident right away, i.e. spelling error or indentation error, I would look at the operation to see if it's the correct one for the intended purpose and whether or not it is formatted properly. If neither of those are the issues, I would then direct my attention to the variables it uses.

The first step in analyzing the variables is to compare the values I'm expecting them to have to the values that would cause the specific crash. For each of these variables, I would trace them back through the code to make sure their values remain valid. I would perform this by adding print statements for the variables before and after they are modified, and by checking return values as well.

This process can be tedious and time consuming, but it will almost certainly help locate the bug in the code. A good practice to make this process easier is to check parameters and return values in all programs when necessary. Additionally, online resources can be very helpful in debugging the code or explaining error messages too.

Deb Packaging details(~ 2 hours):
Resources used:

1. Debian New Maintainers' Guide - for most of the questions.
2. I did use ChatGPT for question 2, since I could not find it in the first resource.

Questions:

1. What do debian/control and debian/rules files do?
    a. The debian/control file acts as an overview of the deb package. In a structured way, the control file provides information of the deb package such as the name, version, maintainer, dependencies, size, description, and more. These values will be used by other package management tools, and is a required file.

b. The debian/rules file is an executable Makefile that is used to create the package. Like other makefiles, it contains targets and instructions to carry them out. Two of the main ones are clean and build used to clean all compile code and to build the package, respectively.

2. What needs to be considered when modifying files not in the debian/* directory
    a. Two things that need to be considered when modifying files not in the debian/* directory are how it affects the behavior of the package and how it can affect developers upstream.
3. What would you need to do to create multiple binary packages out of one source package
    a. The best way is to use wrapper tools, which requires the installation of the devscripts tools. Next you want to get the source package and change to the source tree. Install needed build-dependencies (if there are any), create a dedicated version of your own build, and build the package.
    b. Without devscripts, you will need all *.dsc, *.tar.gz and *.diff.gz files to compile the source. Then you can extract the package into a new version. Go into the new directory and you can compile and build it.
4. What steps would it take to upload that package to Debian or Ubuntu
    a. You'll have to become an official developer. Then you can upload it manually or use existing automatic tools like dupload or dput. With dupload, you'll have to edit the config file, override a few things, then upload using a command.

Git Basics (~ 1 hour):

Resources: Git - Reference (git-scm.com)

1. What does 'git rebase -i' do?
    a. git rebase is a command similar to merge, that takes all of the commits on one branch, and writes them on another. The 'git rebase -i' command creates a temporary text file with options you can pick and choose from to customize the rebase. Upon closing the editor of this text file, the rebasing will commence.
2. How do I switch to a remote branch and track any changes on the remote branch?
    a. git branch -r will display the remote branches available in your repository. From the list of those, you can switch and track the branch with the command 'git checkout -t <remote>/<branch>' where remote and branch are the names of the remote and branch you are looking to track.
3. When you find out the special line which may be the root cause of the bug and want to find out who is the author of the line, what will you do?
    a. The command 'git blame' can help. When you have the name and line of the file in question, you can write 'git blame <file path/name> -L <start line>,<end line>' where file path/name is the file, start line is where you want to start looking and

end line is the end. The two lines will be the same if you want to analyze just one line.

4. When you find a commit which is a possible fix to the bug what will you do?
    a. You should test the commit to see if it can fix the bug. To do this, you should create and switch to a temporary branch copied from the one with the bug. On that branch, 'git cherry-pick' the commit you believe to fix the bug and test it. If it does fix the bug, then you can redo the cherry-pick in the original branch, or merge the temporary branch into the original.

OSS participation and prior art

Code
Point us to, or share with us, code that you are especially proud of, and explain why you're proud of it.

File: GameManager.cs

This file is a C# script for the mobile game I am developing. While there are thousands of lines of code located in other files, this is the largest and most important one. I am proud of this code, and of my game overall because it is something I have put a lot of effort into over the last several months and something I did all by myself. At the beginning of the summer, I had set out to create a mobile game. I took a ~30 hour online course to learn how to create games in Unity and code in C#. The process of building the project took a lot of debugging, searching the Unity API, and referencing stack overflow to get it to be a fun and playable game. This project has taught me many of the fundamentals for project development and drastically improved my ability and confidence to learn and develop programs on my own.

Open Source
If you have participated in open source software previously, please point us to public repos that you have participated in.

- I have not participated in any open source software yet.

Community
If you have common community based accounts like Launchpad, Github, Gitlab, Ask Ubuntu, Stack Overflow, Salsa, ... please point to them here.

- Github: seanh18 (Sean) (github.com) I mostly use it for class projects.

Packaging

If you have done any packaging work (debs, rpms, docker, snaps, pypi, ...) please share any links to packages you've built.

- I use docker to manage a web application I created that searches and displays information in a database, but it is proprietary so I cannot share it.