Requirements:

1. Must be able to trim a sound to a user specified length and have the option of saving as a new file
2. Must be able to reverse the playback direction of sounds and have the option of saving as a new file
3. Users must be able to access sounds from different folders
4. Users must be able to create and delete folders
5. Users must be able to rename existing sounds.
6. Users must be able to add and remove sounds from existing folders.
7. Must be able to merge 2 or more sounds into 1 single sound file and have the option of saving as a new file
8. Users must be able to play a series of one or more sounds.
9. System must validate whether commands issued by the user are functional (won’t throw errors).
10. System must work on Linux, Mac, Windows

Story Map:
https://docs.google.com/spreadsheets/d/1MErfD5ugRthvZ-_9jqCmFLVuMGNB0Y3FiVxmHID3AvY/edit#gid=0

Use Case Name	**Trim a sound**
Summary	Some sound files might be the wrong length or a user only wants a subsnippet of the sound. The user can specify how long of a snippet they would like from a sound. The edited sound automatically plays but can be optionally saved in any folder.
Rationale	An integral part of a sound archive system is to have a method of changing the length of sounds.
Users	The User of the sound archive
Preconditions	The audio archive system is running The file exists in the directory The entered start and end times exist within the confines of the sound Start time <= end time
Course of events	1. The User starts the program by running ‘python main.py’ 2. The user types ‘trim_sound’ <file_name> <start_time> <end_time> [-out=<path_to_new_file>] 3. The do_trim_sound() function is called in main() 4. The trim_sound function is validated 5. The system creates a new .wav file (trimmed.wav) in the directory 6. The system trims trimmed.wav according to the specified times 7. The system plays the edited sound 8. The system renames trimmed.wav to <path_to_new_file> and saves it in the desired directory
Exceptions	If the user doesn’t enter an existing sound, the function errors and the help trim_sound message is displayed. If the user enters a start time greater than the end time, the function errors and the help trim_sound message is displayed. If the user doesn’t enter an end time, the function errors and the help trim_sound message is displayed.
Alternative paths	(2) The user can optionally enter a <path_to_new_file> to save the edited file as a new sound (8) Alternatively, the system deletes ‘trimmed.wav’ if -out= is not included in the command
Postconditions	The trimmed sound is saved as a new independent file if the -out flag is included. The new sound file acts as a normal sound and can be subsequently trimmed and saved again.

Use Case Name	Merging Sounds into a single file.

Summary	While users can play any number of sounds using the play function, they have to type out all of the sounds they want to play each time they want to play them. To allow the user to play a combination of existing sounds by writing only a single sound name, all of the component sounds can be merged into one.
Rationale	To group or share sounds from multiple files, the sounds should be able to be grouped into a single file. These sounds can be edited in combination later if they're saved.
Users	Everyone, especially people interested in sharing sounds or who want to play some set of sounds in a set order repeatedly.
Preconditions	Must have at least one sound, but ideally 2+ sounds. (1 sound allows you to merge the same sound to itself while 2+ sound means that different sounds can be put together).
Course of events	1. The user starts the CLI app by calling 'python main.py'.
	2. The user types 'merge <file_name(s)> [-out=<path_to_new_file>]' a. The user may type 'merge' or 'help merge' to find correct formatting before calling the function.
	3. The 'do_merge()' function in main is called with the additional arguments passed
	4. The merge function is validated by calling 'validate()' within main. a. The 'validate_merge()' function in validate.py is called and returns True if the arguments were valid.
	5. In main, 'do_merge()' calls the implementation of merge in the Effects class.
	6. Each sound passed as an argument is added to a new merged audio file.
	7. The combined sound is exported to a new temporary file in the current working directory.
	8. The temporary combined sound is played.
	9. The temporary sound is either renamed (if the user used the -out flag) or deleted.
Exceptions	If the user doesn't pass enough sounds (1+) to merge, then no sounds are merged and the help for the merge function is displayed instead.
	If the user calls the merge function with a tag other than -out=[filename] (if it is a different flag entirely or if the user didn't include anything after the '=' in the -out flag), then the function errors and doesn't merge the sounds.
	If any of the sounds passed to the function are not valid sounds (if they don't exist or if they aren't of the right file type), then the function errors and doesn't merge the sounds.
	If the new file name (and path) specified in the -out flag already exists, then the combined sound cannot save to that location.
Alternative paths	If the user chooses to not include the -out flag, then the combined sound will play but no combined file will be saved. If the user wants to play the sound after testing how it sounds, the same command could be run again with the -out flag, making it accessible after the sound plays.
Postconditions	The combined sound file is saved to where the user specified in the -out flag. The combined sound can be used as a normal sound throughout the rest of the app.

Use Case Name	Rename

Summary	Users may wish to change the name of the sounds they work with to better organize their directory. The user renames files in response to prompting from the application after inputting the rename option. The renamed file appears in place of the original file.
Rationale	Users should able to identify their sounds however they wish at any time
Users	Everybody who uses the application, people who may have misnamed files on creation or think that another naming scheme will be better.
Preconditions	- The input file exists in the directory - There doesn't already exist a file with the new name in the current folder.
Course of events	1. The user starts the CLI app by calling 'python main.py'.
	2. The user invokes rename by typing 'rename <original_file> <new_name>' a. The user may type 'rename' or 'help rename' to find correct formatting before calling the function.
	3. The do_rename()' function in main is called with the additional arguments passed
	4. The rename function is validated by calling 'validate()' within main. a. The 'validate_rename()' function in validate.py is called and returns True if the arguments were valid.
	5. In main, 'do_rename()' calls the implementation of rename in the FileManager class.
	6. Using the os library, the original file is renamed to the new name passed by the user.
Exceptions	If the user doesn't pass 2 arguments, then there was either insufficient information provided or there was too much for the parser to understand. The function errors and returns.
	If the initial sound file (that the user is trying to rename) doesn't exist, then it causes an error.
	If the new name is linked to a file that already exists, then the sound cannot be renamed to the name of a sound that already exists. This causes an error and the function returns.
Alternative paths	If the user chooses to not include the -out flag, then the combined sound will play but no combined file will be saved. If the user wants to play the sound after testing how it sounds, the same command could be run again with the -out flag, making it accessible after the sound plays.
Postconditions	The combined sound file is saved to where the user specified in the -out flag. The combined sound can be used as a normal sound throughout the rest of the app.

Name	Adding a new folder
Summary	Users might want to add a new folder to their sound library in order to better organize their collection of sounds. After the folder is created, users should be able to add new sounds to it (see other use case,) or move existing sounds into it.
Rationale	An organized sound library is a beautiful sound library. If the user wants to sort sounds on their own, being able to make a new folder gives them another place to store sounds within.
Users	Any user of the audio archive who wishes to organize their sounds in a folder structure
Preconditions	The audio archive program is running.
Course of events	1. The user accesses the CLI 2. User types new_folder <folder_name>. 3. The system validates the folder name for conventions and checks if it already exists. 4. The system creates the folder
Exceptions	- A folder with that name already exists (do not create a new folder) - More than one argument is passed (we only create one folder at a time, do not accept this command) - User tries to create a folder with a name that is invalid on the system. Do not create the folder - User tries to create a folder, but does not have permission to do so. Do not create the folder -
Alternative Paths	N/A
Post conditions	A new, empty, folder exists and is accessible for playback by the program

Use Case Name	**Reverse the Playback Direction of a Sound**
Summary	A conspiracy theorist believes there are secret messages hidden in a sound. They can reverse the playback direction of the sound to check for said messages.
Rationale	To create new ways to listen to sounds, sounds must be able to be played in reverse. If the desired edit is satisfactory, the edited sound can be saved as a new file.
Users	Everyone with access to the sound archive
Preconditions	The audio archive system is running The file exists in the directory FFmpeg is downloaded
Course of events	1. The User starts the program by running ‘python main.py’ 2. The user types ‘reverse’ <file_name> [-out=<path_to_new_file>] 3. The do_reverse() function is called in main() 4. The reverse function is validated 5. The system creates a new .wav file (reverse.wav) in the directory 6. The system reverses reverse.wav 7. The system plays reverse.wav 8. The system renames reverse.wav to <path_to_new_file> and saves it in the desired directory
Exceptions	If the user doesn’t enter an existing sound, the function errors and the help reverse message is displayed. If the user enters more than one sound file, the function errors and the help reverse message is displayed. If the user enters an unrecognized command, the function errors and the commands help message is displayed
Alternative paths	(2) The user can optionally enter -out=<path_to_new_file> to save the edited file as a new sound (8) Alternatively, the system deletes ‘reverse.wav’ if -out= is not included in the command
Postconditions	The reversed sound is saved as a new independent file if the -out flag is included. The new sound file acts as any other existing sound file.

Use Case Name	Adding new sounds to folder.
Summary	If a user has a sound they want to add to a specific folder, they can specify the path to the file and the folder to add the sound to that folder. The sound is copied from its original location on the computer to the folder, meaning the new sound is a copy.
Rationale	Adding sounds to an audio archive is what allows the archive to grow. It isn't convenient to have to exit the CLI and go to the file manager on your computer in order to add new sounds that exist elsewhere on the computer, so this command allows the user to move audio files from within the app. If someone wanted to manually create a playlist, they can add preexisting sounds to a new playlist.
Users	All users, especially those who care about the organization of their sounds or who want to add external sounds into the viewable folders on the app.
Preconditions	There has to be a folder in the current working directory and a path to a valid audio file. The folder cannot already contain an audio file with the same name as the file to be added.
Course of events	1. The user starts the CLI app by calling 'python main.py'
	2. The user types 'add_sound <folder_to_add_to> <path_to_original_file>' into the CLI.
	3. The 'do_add_sound()' function in main is called with the additional arguments passed.
	4. The add_sound function is validated by calling 'validate()' within main. In validation, the 'validate_add_sound()' function returns True if the arguments were valid.
	5. In main, 'do_add_sound()' calls the implementation of add_sound in FileManager.
	6. 'add_sound()' in FileManager gets the path to the source audio file and the target directory, using the shutil library to copy the source audio into the target directory
Exceptions	If the user doesn't pass two arguments, then the command isn't valid and the function doesn't run. If they passed fewer than 2 arguments, then they must be missing either the sound that's being added or the folder the sound is being added to. If the user passed more than 2 arguments, then they typed something beyond the directory and sound which cannot be recognized.
	If the first argument passed to the add_sound function isn't a directory/folder, then an error is thrown and the command doesn't execute.
	If the second argument passed to the add_sound function isn't a valid audio file, then an error is thrown and the command doesn't execute.
Alternative paths	The user may type 'add_sound' or 'help add_sound' to find the correct formatting for the arguments of the function before actually calling the function.
Postconditions	The sound is copied from the original location into the specified folder. The original sound is not removed from its original location.

Use Case Name	Playing Sounds
Summary	If the user has sounds collected, they likely want to be able to play the sounds. There's different ways to play sounds, which are supported as subcalls within the play command.
Rationale	Playing sounds allows users to hear what the sounds are. This can either be the final purpose of the app (for most users), or it can help users sort their files (as they need to play them to know what they are.
Users	Users, or anyone with sounds to play.
Preconditions	The sound(s) to play exist and can be found under the current working directory.
Course of events	1. The user starts the CLI app by calling 'python main.py'.
	2. The user types 'play [-multi|-seq|-rand|-delay={delaytime}] <file_name(s)>'. (if the user types 'play' or 'help play', the help menu displays to show the proper syntax for using the function).
	3. The 'do_play()' function in main is called with the additional arguments passed.
	4. The play function is validated through the 'validate()' call in main that calls 'validate_play()' in validate.py (which returns true if there's no issues with the statement).
	5. The 'play()' function in AudioPlayer is called, which plays sounds sequentially by default.
	6. Each sound passed as an argument is played to completion sequentially.
Exceptions	If an unrecognized flag is used (not -multi, -delay, -mute, -seq, or -rand) then an error is thrown and the sound(s) doesn't play.
	If a non-flag passed argument is not a directory and not a sound, then the validator catches it as an invalid argument and throws and error.
Alternative paths	If the user calls the play function with the -delay=<float> flag, then the audio files play with the start of each sound happening the specified number of seconds after the start of the one before it. (If no float value is provided, then the flag isn't valid and an error is thrown).
	If the user calls the play function with the -mute flag, then nothing plays (because it's muted).
	If the user calls the play function with the -multi flag, then all of the audio files passed as arguments start playing at the same time. The function waits until the last sound finishes playing and returns control back to the CLI.
	If the user calls the play function with the -rand flag and passes a valid folder, then a random sound is chosen from the folder and played. (If the folder isn't valid (like if it's empty), then an error is thrown and the sound doesn't play).
Postconditions	The played sounds play and control is returned to the CLI, ready for another command.