

Sheet Music PyQt5 GUI

Imports

Standard Library

```
In [1]: # Standard Library imports
import re
import os
import sys
import time
from collections import defaultdict
from datetime import datetime
import subprocess
import platform
```

Third-Party Libraries

```
In [2]: # PyQt5
from PyQt5 import QtGui
from PyQt5.QtGui import QIcon, QTextCursor
from PyQt5.QtCore import (Qt, pyqtSlot, QThread, pyqtSignal, QTimer, QMetaType)
from PyQt5.QtWidgets import (QApplication, QMainWindow, QLineEdit, QProgressBar, QCheckBox,
                             QPushButton, QVBoxLayout, QWidget, QLabel, QComboBox, QTextEdit)

# Selenium
from selenium import webdriver
from selenium.webdriver.chrome.service import Service
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
from selenium.common.exceptions import NoSuchElementException, StaleElementReferenceException, T
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

from webdriver_manager.chrome import ChromeDriverManager
import requests
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
from torch.nn.functional import pad
from reportlab.pdfgen import canvas
from reportlab.lib.pagesizes import letter
```

Local Imports

```
In [3]: # Local imports
from model_functions import UNet, VDSR, PIL_to_tensor, tensor_to_PIL, load_best_model
```

Threads

Find Songs Thread

```
In [4]: # Thread to find songs based on the user's search query
class FindSongsThread(QThread):
    progress = pyqtSignal(int)
    status = pyqtSignal(str)
    song_info_updated = pyqtSignal(str)
    song_choice_box_updated = pyqtSignal(str)
    log_updated = pyqtSignal(str)
    clear_song_info = pyqtSignal()
    clear_song_choice_box = pyqtSignal()
    clear_key_choice_box = pyqtSignal()
    request_song_choice_box_count = pyqtSignal()
    receive_song_choice_box_count = pyqtSignal(int)
    insert_separator_in_song_choice_box = pyqtSignal(int)

    def __init__(self, driver, user_song_choice):
        super().__init__()
        self.driver = driver
        self.user_song_choice = user_song_choice
        self.song_choice_box_count = 0
        self.receive_song_choice_box_count.connect(self.set_song_choice_box_count)

    def set_song_choice_box_count(self, count):
        """Set the song choice box count."""
        print(f"Received count in set_song_choice_box_count is: {count}")
        self.song_choice_box_count = count

    def run(self):
        """Run the thread to find songs."""
        print("Starting find_songs function")
        self.status.emit("")
        self.progress.emit(0)
        print(f"user_song_choice is: {self.user_song_choice}")
        print("Clearing song_info")
        self.clear_song_info.emit()
        print("Clearing song_choice_box")
        self.clear_song_choice_box.emit()
        print("Clearing key_choice_box")
        self.clear_key_choice_box.emit()

        url = "https://www.praisecharts.com/search"
        self.driver.get(url)

        # Try to locate and interact with the search bar
        try:
            print("Locating search bar")
            search_xpath = '//*[@id="search-input-wrap"]/input'
            search_bar = WebDriverWait(self.driver, 2).until(
                EC.element_to_be_clickable((By.XPATH, search_xpath))
            )
            print("Clicking search bar")
            search_bar.click()

            # Add an additional check before entering text
            if search_bar.is_enabled() and search_bar.is_displayed():
                print("Clearing search bar")
                search_bar.clear()
                print("Entering user_song_choice into search bar")
                search_bar.send_keys(self.user_song_choice)
```

```

except StaleElementReferenceException:
    print("Search bar StaleElementReferenceException")
    self.log_updated.emit("Stale element reference for search bar.")
    return
except NoSuchElementException:
    print("Search bar NoSuchElementException")
    self.log_updated.emit("Missing element reference for search bar.")
    return
except TimeoutException:
    print("Search bar TimeoutException")
    self.log_updated.emit("Search bar not found.")
    return

songs_counter = 0

try:
    print("Entering song search loop")
    # Find the parent element
    songs_parent_xpath = '//*[@id="page-wrapper"]/ion-router-outlet/app-page-search/ion-
WebDriverWait(self.driver, 2).until(
        EC.presence_of_all_elements_located((By.XPATH, songs_parent_xpath))
    )
    time.sleep(1)
    songs_parent = self.driver.find_element("xpath", songs_parent_xpath)

    # Find all children under the parent element
    songs_children = songs_parent.find_elements("xpath", 'app-product-list-item')

    # Loop through children to get text
    for idx, child in enumerate(songs_children, 1):

        title = ''
        text2 = ''
        text3 = ''

        try:
            title = child.find_element("xpath", './div/a/div/h5').text
        except NoSuchElementException:
            pass

        try:
            text3 = child.find_element("xpath", './div/a/div/span/span').text
        except NoSuchElementException:
            pass

        if text3 != '':
            try:
                text2 = child.find_element("xpath", './div/a/div/span').text.split("\n")
            except NoSuchElementException:
                pass

        if text3 == text2:
            text2 = ''

        print(f"title is: {title}")
        print(f"text2 is: {text2}")
        print(f"text3 is: {text3}")

        # Concatenate the text and print
        element_text = f"{title}\n{text2}\n{text3}"
        print(f"element_text is: {element_text}")

```

```

        cleaned_text = f"{title} {text2} {text3}"
        print(f"cleaned_text is: {cleaned_text}")

        # Check if the text is empty or None
        if element_text == '':
            break

        # Check if 'keys:' exists in subcategories
        if text3 != '':
            print(f"Adding '{cleaned_text}' to self.song_info")
            self.song_info_updated.emit(element_text)
            print(f"Adding '{cleaned_text}' to song_choice_box")
            self.song_choice_box_updated.emit(element_text)
            songs_counter += 1

            self.request_song_choice_box_count.emit()
            # Insert a separator after the item
            index_to_insert_separator = self.song_choice_box_count+1 # Gets current num
            print(f"Inserting separator for song_choice_box at {index_to_insert_separator}")
            self.insert_separator_in_song_choice_box.emit(index_to_insert_separator)
        else:
            # Only print the title (assuming the title is the first line of the element)
            print(f"Emitting '{title}' to song_info_updated")
            self.song_info_updated.emit(title)

    except StaleElementReferenceException:
        print("Finding songs StaleElementReferenceException")
        self.log_updated.emit("Stale element reference for elements.")
        return
    except NoSuchElementException:
        print("Finding songs NoSuchElementException")
        self.log_updated.emit("Missing element reference for elements.")
        return
    except TimeoutException:
        print("Finding songs TimeoutException")
        self.log_updated.emit("No elements found.")
        return

    self.log_updated.emit(f"Found {songs_counter} songs for search: {self.user_song_choice}")
    print("Stopping find_songs function")

```

Select Song Thread

```

In [5]: # Thread to select a song based on the user's choice
class SelectSongThread(QThread):
    progress = pyqtSignal(int)
    status = pyqtSignal(str)
    button_elements_signal = pyqtSignal(list)
    log_updated = pyqtSignal(str)
    clear_key_choice_box = pyqtSignal()
    key_choice_box_updated = pyqtSignal(str)
    clear_button_elements = pyqtSignal()

    def __init__(self, driver, selected_song, selected_song_index, selected_song_title, user_song):
        super().__init__()
        self.driver = driver
        self.selected_song = selected_song
        self.user_song_choice = user_song_choice
        self.selected_song_index = selected_song_index
        self.selected_song_title = selected_song_title

```

```

def run(self):
    """Run the thread to select a song."""
    print("Starting SelectSongThread run")
    self.log_updated.emit(f"Selected song: {self.selected_song_title}")

    while True:
        try:
            print(f"Waiting to click song index {self.selected_song_index+1}")
            click_xpath = f'//*[@id="page-wrapper"]/ion-router-outlet/app-page-search/ion-co
            click_element = WebDriverWait(self.driver, 2).until(
                EC.element_to_be_clickable((By.XPATH, click_xpath))
            )
            print(f"Clicking song index {self.selected_song_index+1}")
            click_element.click()
            break
        except StaleElementReferenceException:
            print("Song StaleElementReferenceException")
            self.log_updated.emit("Stale element reference for the song.")
            return
        except NoSuchElementException:
            print("Song NoSuchElementException")
            self.log_updated.emit("No element reference found for the song.")
            return
        except TimeoutException:
            print("Song TimeoutException")
            self.log_updated.emit("No song found.")

            print(f"user_song_choice is: {self.user_song_choice}")
            print("Clearing button_elements")
            self.clear_button_elements.emit()
            url = "https://www.praisecharts.com/search"
            self.driver.get(url)

            # Try to locate and interact with the search bar
            try:
                print("Locating search bar")
                search_xpath = '//*[@id="search-input-wrap"]/input'
                search_bar = WebDriverWait(self.driver, 2).until(
                    EC.element_to_be_clickable((By.XPATH, search_xpath))
                )
                print("Clicking search bar")
                search_bar.click()

                # Add an additional check before entering text
                if search_bar.is_enabled() and search_bar.is_displayed():
                    print("Clearing search bar")
                    search_bar.clear()
                    print("Entering user_song_choice into search bar")
                    search_bar.send_keys(self.user_song_choice)

            except StaleElementReferenceException:
                print("Search bar StaleElementReferenceException")
                self.log_updated.emit("Stale element reference for search bar.")
                return
            except NoSuchElementException:
                print("Search bar NoSuchElementException")
                self.log_updated.emit("Missing element reference for search bar.")
                return
            except TimeoutException:
                print("Search bar TimeoutException")
                self.log_updated.emit("Search bar not found.")

```

return

```
try:
    print("Waiting to click 'Chords & Lyrics' button")
    chords_click_xpath = '//*[@id="page-wrapper"]/ion-router-outlet/app-product-page/ion-
chords_click_element = WebDriverWait(self.driver, 2).until(
    EC.element_to_be_clickable((By.XPATH, chords_click_xpath))
)
    print("Clicking 'Chords & Lyrics' button")
    chords_click_element.click()
    print("Successfully clicked 'Chords & Lyrics' button")
except StaleElementReferenceException:
    print("Chords & Lyrics StaleElementReferenceException")
    self.log_updated.emit("Stale element reference for this sheet music.")
    return
except NoSuchElementException:
    print("Chords & Lyrics NoSuchElementException")
    self.log_updated.emit("No element reference found for this sheet music.")
    return
except TimeoutException:
    print("Chords & Lyrics TimeoutException")
    self.log_updated.emit("No sheet music found.")
    return
```

Selecting "Orchestration" button

```
try:
    print("Waiting to click 'Orchestration' button")
    orch_click_xpath = "//h3[contains(text(), 'Orchestration')]/ancestor::div[4]"
    orch_click_element = WebDriverWait(self.driver, 2).until(
    EC.element_to_be_clickable((By.XPATH, orch_click_xpath))
)
    print("Clicking 'Orchestration' button")
    orch_click_element.click()
    self.log_updated.emit("Orchestration found.")
except StaleElementReferenceException:
    print("Orchestration StaleElementReferenceException")
    self.log_updated.emit("Stale element reference for orchestration.")
    return
except NoSuchElementException:
    print("Orchestration NoSuchElementException")
    self.log_updated.emit("No element reference found for orchestration.")
    return
except TimeoutException:
    print("Orchestration TimeoutException")
    self.log_updated.emit("No orchestration found.")
    return
```

Click key menu

```
try:
    print("Waiting to click 'key' menu button")
    key_click_xpath = '//*[@id="page-wrapper"]/ion-router-outlet/app-product-page/ion-co
key_click_element = WebDriverWait(self.driver, 2).until(
    EC.element_to_be_clickable((By.XPATH, key_click_xpath))
)
    print("Clicking 'key' menu button")
    key_click_element.click()

    key_parent_xpath = '//*[@id="page-wrapper"]/ion-router-outlet/app-product-page/ion-co
    key_parent_element = self.driver.find_element("xpath", key_parent_xpath)
    print("Saving individual buttons under key parent")
    button_elements = key_parent_element.find_elements(by=By.TAG_NAME, value='button')
except StaleElementReferenceException:
```

```

        print("Key menu StaleElementReferenceException")
        self.log_updated.emit("Stale element reference for the key menu.")
        return
    except NoSuchElementException:
        print("Key menu NoSuchElementException")
        self.log_updated.emit("No element reference found for the key menu.")
        return
    except TimeoutException:
        print("Key menu TimeoutException")
        self.log_updated.emit("No key menu found.")
        return

    keys = []

    self.clear_key_choice_box.emit()

    try:
        print("Saving buttons in button_elements")
        for button in button_elements:
            keys.append(button.text)
            print(f"Added {button.text} to keys")
            self.key_choice_box_updated.emit(button.text)
            print(f"Added {button.text} to key_choice_box")
    except UnboundLocalError:
        print("button_elements UnboundLocalError")
        self.log_updated.emit("No key menu found.")
        return

    # Click the last button
    if button_elements: # Check to make sure the list is not empty
        first_button = button_elements[0]
        print(f"Clicked the first button: {first_button.text}")
        first_button.click()

    formatted_keys = ', '.join(keys)
    print(f"formatted_keys is: {formatted_keys}")
    self.log_updated.emit(f"Found keys: {formatted_keys}")
    self.log_updated.emit(f"Automatically selected key: {keys[0]}")
    print(f"button_elements about to be emitted are: {button_elements}")
    print("Emitting signal for button_elements")
    self.button_elements_signal.emit(button_elements)
    print("Successfully emitted signal for button_elements")
    print("Stopping SelectSongThread run")

```

Select Key Thread

```

In [6]: # Thread to select a key for the chosen song
class SelectKeyThread(QThread):
    progress = pyqtSignal(int)
    status = pyqtSignal(str)
    log_updated = pyqtSignal(str)

    def __init__(self, driver, selected_key, button_elements):
        super().__init__()
        self.driver = driver
        self.selected_key = selected_key
        self.button_elements = button_elements.copy() # make a copy to avoid modifying the shared

    def run(self):
        """Run the thread to select a key."""

```

```

try:
    print("Attempting to click 'key' menu button first")
    key_click_xpath = '//*[@id="page-wrapper"]/ion-router-outlet/app-product-page/ion-co
    key_click_element = WebDriverWait(self.driver, 2).until(
        EC.element_to_be_clickable((By.XPATH, key_click_xpath))
    )
    print("Clicking 'key' menu button")
    key_click_element.click()
except NoSuchElementException:
    print("Key menu NoSuchElementException (Before entering loop)")

print(f"self.button_elements at the beginning is: {self.button_elements}")

button_clicked = False # A flag to indicate if a button was clicked

while not button_clicked:
    for button in self.button_elements:
        print(f"button.text is: {button.text}")
        if self.selected_key == button.text:
            button.click()
            button_clicked = True # Set the flag to True
            break

    if not button_clicked: # Check if no button was clicked
        try:
            print("Attempting to click 'key' menu button again")
            key_click_xpath = '//*[@id="page-wrapper"]/ion-router-outlet/app-product-page
            key_click_element = WebDriverWait(self.driver, 2).until(
                EC.element_to_be_clickable((By.XPATH, key_click_xpath))
            )
            print("Clicking 'key' menu button")
            key_click_element.click()
        except NoSuchElementException:
            print("Key menu NoSuchElementException (Inside loop)")

print(f"self.button_elements at the end is: {self.button_elements}")

self.log_updated.emit(f"Selected key: {self.selected_key}")

```

Download and Process Thread

```

In [7]: # Thread to download and process images based on the selected song and key
class DownloadAndProcessThread(QThread):
    progress = pyqtSignal(int)
    status = pyqtSignal(str)
    log_updated = pyqtSignal(str)

    def __init__(self, driver, key_choice_text, selected_song_title, selected_song_artist, paths):
        super().__init__()
        self.driver = driver
        self.key_choice_text = key_choice_text
        self.selected_song_title = selected_song_title
        self.selected_song_artist = selected_song_artist
        self.paths = paths
        self.download_horn_only = download_horn_only

    def run(self):
        """Run the thread to download and process images."""
        print("Running self.initialize_directories()")
        song_dir, temp_dir = self.initialize_directories()

```



```

print("Running find_parts()")
self.find_parts()
print("Running self.download_images(temp_dir)")
self.download_images(temp_dir)
print("Running self.remove_watermarks()")
self.remove_watermarks()
print("Running self.upscale_images()")
self.upscale_images()
print("Emptying cuda cache")
torch.cuda.empty_cache()
print("Running self.create_pdfs(song_dir, temp_dir)")
self.create_pdfs(song_dir, temp_dir)
print("Running self.cleanup")
self.cleanup(temp_dir)
print(f"Opening directory {song_dir}")
self.open_directory(song_dir)

def initialize_directories(self):
    """Initialize directories for saving downloaded images and temporary files."""
    print(f"Saving {self.key_choice_text} as key_dir")
    key_dir = self.key_choice_text

    title_dir = re.sub(r'[<>:"\\|?* ]', '_', self.selected_song_title.replace("/", "-"))
    print(f"title_dir is: {title_dir}")
    artist_dir = re.sub(r'[<>:"\\|?* ]', '_', self.selected_song_artist.replace("/", "-"))
    print(f"artist_dir is: {artist_dir}")

    main_dir = self.paths['download_dir']
    title_sub_dir = title_dir
    artist_sub_dir = artist_dir
    temp_sub_dir = self.paths['temp_sub_dir']

    song_dir = os.path.join(main_dir, title_sub_dir, artist_sub_dir, key_dir)
    os.makedirs(song_dir, exist_ok=True)
    print(f"Making song_dir {song_dir}")

    temp_dir = os.path.join(main_dir, title_sub_dir, artist_sub_dir, temp_sub_dir)
    os.makedirs(temp_dir, exist_ok=True)
    print(f"Making temp_dir {temp_dir}")
    return song_dir, temp_dir

def find_parts(self):
    """Find the available parts for the selected song."""
    try:
        click_xpath = '//*[@id="page-wrapper"]/ion-router-outlet/app-product-page/ion-content'
        click_element = WebDriverWait(self.driver, 2).until(
            EC.element_to_be_clickable((By.XPATH, click_xpath))
        )
        print("Opening parts menu")
        click_element.click()

        parent_element = self.driver.find_element("xpath", '//*[@id="page-wrapper"]/ion-router-outlet/app-product-page/ion-content')
        parts_elements = parent_element.find_elements(by=By.TAG_NAME, value='button')
    except StaleElementReferenceException:
        print("Parts menu StaleElementReferenceException")
        self.log_updated.emit("Stale element reference for the key menu.")
        return
    except NoSuchElementException:
        print("Parts menu NoSuchElementException")
        self.log_updated.emit("No element reference found for the key menu.")
        return
    except TimeoutException:

```

```

        print("Parts menu TimeoutException")
        self.log_updated.emit("No key menu found.")
        return

self.instrument_parts = []

try:
    print("Saving buttons in parts_elements")
    for part in parts_elements:
        if 'cover' not in part.text.lower():
            self.instrument_parts.append(part.text)
            print(f"Added {part.text} to self.instrument_parts")
except UnboundLocalError:
    print("button_elements UnboundLocalError")
    self.log_updated.emit("No parts menu found.")
    return

print(f"self.instrument_parts is: {self.instrument_parts}")
print(f"self.instrument_parts len is: {len(self.instrument_parts)}")

try:
    click_xpath = '//*[@id="page-wrapper"]/ion-router-outlet/app-product-page/ion-content
click_element = WebDriverWait(self.driver, 2).until(
    EC.element_to_be_clickable((By.XPATH, click_xpath))
)
    print("Closing parts menu")
    click_element.click()
except StaleElementReferenceException:
    print("Parts menu StaleElementReferenceException")
    self.log_updated.emit("Stale element reference for the key menu.")
    return
except NoSuchElementException:
    print("Parts menu NoSuchElementException")
    self.log_updated.emit("No element reference found for the key menu.")
    return
except TimeoutException:
    print("Parts menu TimeoutException")
    self.log_updated.emit("No key menu found.")
    return

def download_images(self, temp_dir):
    """Download images of the sheet music."""
    self.full_paths = []
    downloaded_files = set() # To keep track of downloaded files
    image_xpath = '//*[@id="preview-sheets"]/div/div[1]/div/img'
    button_xpath = "//button[contains(@class, 'sheet-nav-gradient-button-right')]" # Replace
    index = 1
    last_instrument_and_song = None # To keep track of the last instrument and song name cor

    while True:
        try:
            # Locate all image elements
            print("Locating image element")
            image_element = WebDriverWait(self.driver, 2).until(
                EC.presence_of_element_located((By.XPATH, image_xpath))
            )
        except StaleElementReferenceException:
            print("Image download StaleElementReferenceException")
            self.log_updated.emit("Image stale element reference found.")
            break
        except NoSuchElementException:
            print("Image download NoSuchElementException")

```

```

self.log_updated.emit("Image no element reference found.")
break
except TimeoutException:
    print("Image download TimeoutException")
    self.log_updated.emit("No more images found")
    break
except Exception as e:
    print(f"Image download {e}")
    self.log_updated.emit(f"An unexpected error occurred when finding image: {e}")
    break

# Rest of the code remains similar
image_url = image_element.get_attribute('src')
print(f"Getting image url: {image_url}")

basename = os.path.basename(image_url)
print(f"basename is: {basename}")
basename_without_page = "_".join(basename.split("_")[:-1])
print(f"basename_without_page is: {basename_without_page}")

self.status.emit(f"Downloading {basename}")

if 'cover' not in image_url:
    if basename_without_page != last_instrument_and_song:
        progress_value = int((index / len(self.instrument_parts)) * 100)
        self.progress.emit(progress_value)
        index += 1

last_instrument_and_song = basename_without_page
print(f"last_instrument_and_song is: {last_instrument_and_song}")

try:
    # --- Modification: Check if "horn" is in the image URL or filename ---
    if self.download_horn_only and "horn" not in image_url.lower():
        print("Skipping image as it does not contain the word 'horn'")
        button_element = self.driver.find_element("xpath", button_xpath)
        button_element.click()
        continue

    # Check if the word 'cover' is in the image URL
    if 'cover' in image_url:
        self.log_updated.emit("Skipping cover image")
        button_element = self.driver.find_element("xpath", button_xpath)
        button_element.click()
        continue

    image_file_name = os.path.basename(image_url)
    full_path = os.path.join(temp_dir, image_file_name)

    # Check if the file already exists
    if os.path.exists(full_path):
        print("File already exists, skipping download")
        self.full_paths.append(full_path)
        button_element = self.driver.find_element("xpath", button_xpath)
        button_element.click()
        continue

    # Skip if the file name is duplicate (in the set)
    if image_file_name in downloaded_files:
        print("Skipping duplicate file")
        self.log_updated.emit(f"Skipping duplicate image {image_file_name}")
        button_element = self.driver.find_element("xpath", button_xpath)

```

```

        button_element.click()
        continue

    self.full_paths.append(full_path)
    downloaded_files.add(image_file_name) # Add the name to the set of downloaded f

    # Download the image using requests
    print(f"Downloading image at {image_url}")
    response = requests.get(image_url)
    if response.status_code == 200:
        with open(full_path, 'wb') as f:
            f.write(response.content)

    print("Clicking next song button")
    # For the button element
    button_element = WebDriverWait(self.driver, 2).until(
        EC.element_to_be_clickable((By.XPATH, button_xpath))
    )
    button_element.click()

except StaleElementReferenceException:
    print("Image download StaleElementReferenceException")
    self.log_updated.emit("Next button element reference stale.")
    break
except NoSuchElementException:
    print("Image download NoSuchElementException")
    self.log_updated.emit("Next button element reference not found.")
    break
except TimeoutException:
    print("Image download TimeoutException")
    self.log_updated.emit("Next button not found")
    break
except Exception as e:
    print(f"Image download {e}")
    self.log_updated.emit(f"An unexpected error occurred clicking next button: {e}")
    break

self.log_updated.emit("Downloaded images")

print("Creating full_path_names")
self.full_path_names = []
for path in self.full_paths:
    print(f"Adding {path} to full_path_names")
    self.full_path_names.append(os.path.basename(path))

def remove_watermarks(self):
    """Remove watermarks from the downloaded images."""
    print("Initializing model wm_model")
    self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    wm_model = UNet().to(self.device)
    load_best_model(wm_model, self.paths['wm_model_path'])
    wm_model.eval()

    print("Initializing images list")
    self.images = []
    for idx, path in enumerate(self.full_paths):
        print(f"Adding {path} image to images")
        self.images.append(PIL_to_tensor(path).unsqueeze(0).to(self.device))

    print("Initializing wm_outputs list")
    self.wm_outputs = []
    self.status.emit("Removing watermarks")

```

```

self.progress.emit(0)

with torch.inference_mode():
    # Zip wm_outputs and full_path_names together
    for idx, (image, full_path_name) in enumerate(zip(self.images, self.full_path_names)):
        print(f"Removing watermark from image {idx}")
        wm_output = wm_model(image)
        self.wm_outputs.append(wm_output)

        # Emit the name of the PNG file for which the watermark is being removed
        self.status.emit(f"Unwatermarking {os.path.basename(full_path_name)}")

        progress_value = int(((idx + 1) / len(self.images)) * 100)
        self.progress.emit(progress_value)

self.log_updated.emit("Removed watermarks")

def upscale_images(self):
    """Upscale the watermarked-removed images."""
    total_images = len(self.wm_outputs)

    print(f"Initializing model us_model, total images to process: {total_images}")

    # Upscaling
    image_base_width = 1700
    image_base_height = 2200

    print("Initializing model us_model")
    us_model = VDSR().to(self.device)
    load_best_model(us_model, self.paths['us_model_path'])
    us_model.eval()

    print("Setting resolutions to 1700x2200")
    upsample = nn.Upsample(size=(image_base_height, image_base_width), mode='nearest')
    wm_outputs_upscaled = []
    self.status.emit("Changing image resolution")
    self.progress.emit(0)
    for idx, wm_output in enumerate(self.wm_outputs):
        print(f"Changing image resolution for image {idx+1} of {total_images}")
        wm_outputs_upscaled.append(upsample(wm_output))
        progress_value = int(((idx+1) / len(self.wm_outputs)) * 100)
        self.progress.emit(progress_value)

    self.log_updated.emit(f"Changed resolutions")

    patch_height = 550
    patch_width = 850
    padding_size = 16

    patch_num = (image_base_width / patch_width) * (image_base_height / patch_height)

    # Total number of patches for all images
    total_patches = total_images * patch_num

    # Counter for the total patches processed so far
    total_patches_processed = 0

    print("Initializing us_outputs list")
    self.us_outputs = []
    self.status.emit("Upscaling images")
    self.progress.emit(0)

```

```

with torch.inference_mode():
    for idx, (wm_output_upscaled, full_path_name) in enumerate(zip(wm_outputs_upscaled,
                                                                    self.full_path_names)):
        print(f"Upscaling image {idx+1} of {total_images}")

        padding = (padding_size, padding_size, padding_size, padding_size)
        wm_output_upscaled_padded = pad(wm_output_upscaled, padding, value=1.0) # Assume black

        # Initialize tensor to store the predicted upscaled image
        us_output = torch.zeros_like(wm_output_upscaled).cpu() # Create CPU tensor here

        self.status.emit(f"Upscaling {os.path.basename(full_path_name)}")

        # Iterate over the specified pixel patches
        for i in range(0, wm_output_upscaled.shape[-2], patch_height):
            for j in range(0, wm_output_upscaled.shape[-1], patch_width):
                print(f"Processing patch with top-left corner at ({i}, {j}) for image {idx+1} of {total_images}")
                patch = wm_output_upscaled_padded[:, :, i:i+patch_height+padding_size*2, j:j+patch_width+padding_size*2]

                # Pass through the model
                us_patch = us_model(patch.to(self.device))

                # Remove the extra padding from each edge of the predicted patch
                us_patch = us_patch[:, :, padding_size:-padding_size, padding_size:-padding_size]

                # Place the predicted patch into the correct location
                us_output[:, :, i:i+patch_height, j:j+patch_width] = us_patch.cpu()

                # Increment the counter for total patches processed
                total_patches_processed += 1

                # Update the progress value based on total patches processed
                progress_value = int((total_patches_processed / total_patches) * 100)
                self.progress.emit(progress_value)

        self.us_outputs.append(us_output)

    print(f"Finished processing image {idx+1} of {total_images}")

self.log_updated.emit("Upscaled images")

def create_pdfs(self, song_dir, temp_dir):
    """Create PDFs from the upscaled images."""
    # Initialize dictionary to group images by instrument
    images_by_instrument = defaultdict(list)

    print("Creating PDFs")
    total_images = len(self.us_outputs) # Total number of images to be processed
    total_images_processed = 0 # Counter to keep track of total images processed

    # Assume `self.us_outputs` is a list of image tensors, and `file_names` is a list of corresponding file names
    for image_tensor, file_name in zip(self.us_outputs, self.full_path_names):
        print(f"Creating groups for {file_name}")
        pdf_name_prefix = "_".join(file_name.split("_")[:-1]) # Extract the instrument name
        images_by_instrument[pdf_name_prefix].append(image_tensor) # Group by instrument

    # Define image size
    img_width, img_height = 1700, 2200

    # Loop through each instrument and create a PDF
    for instrument_name, image_tensors in images_by_instrument.items():
        print(f"Creating PDF for {instrument_name}")
        pdf_path = f"{instrument_name}.pdf"

```

```

pdf_path = os.path.join(song_dir, pdf_path)
c = canvas.Canvas(pdf_path, pagesize=(img_width, img_height))

self.status.emit(f"Creating {instrument_name}.pdf")

# Loop through each tensor, convert to a PIL Image, and add to the PDF
for idx, image_tensor in enumerate(image_tensors):
    print(f"Adding tensor {idx} to PDF")

    # Update the counter for total images processed and emit progress
    total_images_processed += 1
    progress_value = int((total_images_processed / total_images) * 100)
    self.progress.emit(progress_value)

    image_pil = tensor_to_PIL(image_tensor) # Assume tensor_to_PIL is a function that

    # Save the PIL image to a temporary file
    temp_path = os.path.join(temp_dir, f"temp_image_{idx}.png")
    print(f"Creating temp image {idx} at {temp_path}")
    image_pil.save(temp_path)

    # Add the image to the PDF
    c.drawImage(temp_path, 0, 0, width=img_width, height=img_height)
    c.showPage()

    # Remove the temporary image file
    print(f"Removing temp image {idx} at {temp_path}")
    os.remove(temp_path)

# Save the PDF
c.save()

print("All PDFs created and saved")
self.status.emit(f"Process completed for {self.selected_song_title}")
self.progress.emit(0)
self.log_updated.emit("Processed images")

def cleanup(self, temp_dir):
    """Cleanup temporary files and directories."""
    for path in self.full_paths:
        print(f"Removed path {path}")
        os.remove(path)
    os.rmdir(temp_dir)
    print(f"Removed dir {temp_dir}")
    self.log_updated.emit("Removed temp files")
    self.log_updated.emit(f"Process completed for song: {self.selected_song_title}")

def open_directory(self, path):
    """Open a directory in the file explorer."""
    if platform.system() == "Windows":
        subprocess.run(['explorer', path.replace('/', '\\')])
    elif platform.system() == "Darwin":
        subprocess.run(['open', path])
    elif platform.system() == "Linux":
        subprocess.run(['xdg-open', path])
    else:
        print(f"Unsupported platform: {platform.system()}")

```

Main App

```

In [8]: # Define the main application class
class App(QMainWindow):
    def __init__(self, *args, **kwargs):
        super(App, self).__init__(*args, **kwargs)

        # Paths for resources and models
        self.paths = {
            'window_icon_path': 'data/Church_Music_Watermark/praisecharts-logo-icon-only.png',
            'wm_model_path': 'models/Watermark_Removal',
            'us_model_path': 'models/VDSR',
            'download_dir': 'Praise_Charts',
            'temp_sub_dir': 'temp',
            'tensor_path': 'data/Church_Music_Watermark/mask.png'
        }

        # Set the window icon
        self.setWindowIcon(QIcon(self.paths['window_icon_path']))

        # Initialize the Chrome driver
        options = Options()
        options.add_argument("--start-maximized")
        self.driver = webdriver.Chrome(service=Service(ChromeDriverManager().install()), options=options)
        url = "https://www.praisecharts.com/"
        self.driver.get(url)
        self.song_info = []
        self.button_elements = []
        self.full_paths = []

        # GUI Elements Initialization
        self.init_ui()

    def init_ui(self):
        """Initialize the GUI elements and layout."""
        print("Creating GUI elements")

        # Create a song search box
        self.song_search_box = QLineEdit(self)
        self.song_search_box.setPlaceholderText("Enter the song you're looking for")

        # Create a search button
        self.search_button = QPushButton("Search", self)
        self.search_button.clicked.connect(self.find_songs)

        # Create a song choice box
        self.song_choice_box = QComboBox(self)
        self.song_choice_box.setFixedHeight(45)
        self.song_choice_box.setMaxVisibleItems(100)

        # Create a song choice label
        self.song_label = QLabel("No song selected", self)

        # Create a song select button
        self.song_select_button = QPushButton("Select song", self)
        self.song_select_button.clicked.connect(self.select_song)

        # Create a key choice ComboBox
        self.key_choice_box = QComboBox(self)
        self.key_choice_box.setMaxVisibleItems(100)

        # Create a key choice label
        self.key_label = QLabel("No key selected", self)

```



```

# Create a key select button
self.key_select_button = QPushButton("Select key", self)
self.key_select_button.clicked.connect(self.select_key)

self.horn_checkbox = QCheckBox("Download horn image only", self)

# Create a button to download and process images
self.download_and_process_button = QPushButton("Download/process images", self)
self.download_and_process_button.clicked.connect(self.download_and_process_images)

# Create a Log area
self.log_area = QTextEdit(self)
self.log_area.setReadOnly(True)

# Connect slot to signal
self.song_choice_box.currentIndexChanged.connect(self.check_song_selection)
self.key_choice_box.currentIndexChanged.connect(self.check_key_selection)

self.progress_label = QLabel("", self)
self.progress_label.setText("")
self.progressBar = QProgressBar(self)
self.progressBar.setValue(0)

self.setWindowTitle("Praise Charts Music Downloader")
self.setMinimumWidth(400)
self.setMaximumWidth(800)
self.setMinimumHeight(600)
self.setMaximumHeight(1000)

# Applying Custom Theme
self.setStyleSheet("""
    QWidget {
        background-color: #fafafa;
    }
    QLineEdit, QComboBox, QTextEdit, QProgressBar {
        background-color: white;
        color: #444444;
        border: 1px solid #cccccc;
        border-radius: 4px;
    }
    QPushButton {
        background-color: #ed3124;
        color: white;
        border-radius: 4px;
        padding: 5px 20px;
        min-height: 20px;
    }
    QPushButton:hover {
        background-color: #c7271d;
    }
    QPushButton:disabled {
        background-color: #AAAAAA; /* Gray out when disabled */
    }
    QLabel {
        color: #333333;
        font-weight: bold;
    }
    QProgressBar {
        border: 1px solid #cccccc;
        border-radius: 4px;
        text-align: center;

```

```

        font-weight: bold; /* Bold text */
    }
    QProgressBar::chunk {
        background-color: #ed3124;
    }
    """
)

# Create a layout
print("Adding GUI elements to GUI")
layout = QVBoxLayout()
layout.addWidget(QLabel("Enter song:"))
layout.addWidget(self.song_search_box)
layout.addWidget(self.search_button)

layout.addSpacing(20)

layout.addWidget(QLabel("Choose a song:"))
layout.addWidget(self.song_choice_box)
layout.addWidget(self.song_label)
layout.addWidget(self.song_select_button)

layout.addSpacing(20)

layout.addWidget(QLabel("Choose a key:"))
layout.addWidget(self.key_choice_box)
layout.addWidget(self.key_label)
layout.addWidget(self.key_select_button)

layout.addSpacing(20)

layout.addWidget(self.horn_checkbox)
layout.addWidget(self.download_and_process_button)
layout.addWidget(QLabel("Log:"))
layout.addWidget(self.log_area)
layout.addWidget(self.progress_label)
layout.addWidget(self.progressBar)

# Create a central widget for the main window
central_widget = QWidget()
central_widget.setLayout(layout)
self.setCentralWidget(central_widget)

def append_log(self, message):
    """Append a log message with a timestamp to the log area."""
    timestamp = datetime.now().strftime('%H:%M:%S')
    formatted_message = f"{timestamp}: {message}"
    print(f"Adding '{formatted_message}' to log")
    self.log_area.append(formatted_message)

# Auto-scroll to the bottom
cursor = self.log_area.textCursor()
cursor.movePosition(QTextCursor.End)
self.log_area.setTextCursor(cursor)

def check_song_selection(self, index):
    """Check if a song is selected and update the song label."""
    if index == -1 or self.song_choice_box.currentText() == "":
        self.song_label.setText("No song selected")
    else:
        selected_song = self.song_choice_box.currentText().split('\n')[0]
        self.song_label.setText(f"You selected: {selected_song}")

```

```

def check_key_selection(self, index):
    """Check if a key is selected and update the key label."""
    if index == -1 or self.key_choice_box.currentText() == "":
        self.key_label.setText("No key selected")
    else:
        selected_key = self.key_choice_box.currentText()
        self.key_label.setText(f"You selected: {selected_key}")

def updateProgressBar(self, val):
    """Update the progress bar value."""
    print(f"Updating progress bar with val {val}")
    self.progressBar.setValue(val)

def updateStatusLabel(self, message):
    """Update the status label text."""
    print(f"Updating status label with label {message}")
    self.progress_label.setText(message)

def closeEvent(self, event):
    """Handle the close event to properly close the WebDriver session."""
    print("Closing driver")
    self.driver.close() # Close the Selenium WebDriver session
    event.accept() # Let the window close

@pyqtSlot()
def lock_inputs(self):
    """Lock input elements to prevent user interaction during processing."""
    self.song_search_box.setEnabled(False)
    self.search_button.setEnabled(False)
    self.song_choice_box.setEnabled(False)
    self.song_select_button.setEnabled(False)
    self.key_choice_box.setEnabled(False)
    self.key_select_button.setEnabled(False)
    self.download_and_process_button.setEnabled(False)
    self.horn_checkbox.setEnabled(False)

@pyqtSlot()
def unlock_inputs(self):
    """Unlock input elements after processing is complete."""
    self.song_search_box.setEnabled(True)
    self.search_button.setEnabled(True)
    self.song_choice_box.setEnabled(True)
    self.song_select_button.setEnabled(True)
    self.key_choice_box.setEnabled(True)
    self.key_select_button.setEnabled(True)
    self.download_and_process_button.setEnabled(True)
    self.horn_checkbox.setEnabled(True)

@pyqtSlot()
def find_songs(self):
    """Start the thread to find songs based on the user's search query."""
    print("Starting find_songs function")

    driver = self.driver
    user_song_choice = self.song_search_box.text()

    self.find_songs_thread = FindSongsThread(driver, user_song_choice)
    self.find_songs_thread.log_updated.connect(self.update_log)
    self.find_songs_thread.progress.connect(self.updateProgressBar)
    self.find_songs_thread.status.connect(self.updateStatusLabel)
    self.find_songs_thread.request_song_choice_box_count.connect(self.send_song_choice_box_count)
    self.find_songs_thread.insert_separator_in_song_choice_box.connect(self.insert_separator)

```

```

self.find_songs_thread.clear_song_info.connect(self.clear_song_info)
self.find_songs_thread.clear_song_choice_box.connect(self.clear_song_choice_box)
self.find_songs_thread.clear_key_choice_box.connect(self.clear_key_choice_box)
self.find_songs_thread.song_info_updated.connect(self.update_song_info)
self.find_songs_thread.song_choice_box_updated.connect(self.update_song_choice_box)
self.find_songs_thread.started.connect(self.lock_inputs)
self.find_songs_thread.finished.connect(self.unlock_inputs)
self.find_songs_thread.start()
print("Stopping find_songs function")

@pyqtSlot()
def select_song(self):
    """Start the thread to select a song based on the user's choice."""
    print("Starting select_song function")

    driver = self.driver
    selected_song = self.song_choice_box.currentText()
    selected_song_index = self.song_info.index(selected_song)
    selected_song_title = self.song_choice_box.currentText().split('\n')[0]
    user_song_choice = self.song_search_box.text()
    self.select_song_thread = SelectSongThread(driver, selected_song, selected_song_index, self)
    self.select_song_thread.log_updated.connect(self.update_log)
    self.select_song_thread.progress.connect(self.updateProgressBar)
    self.select_song_thread.status.connect(self.updateStatusLabel)
    self.select_song_thread.key_choice_box_updated.connect(self.update_key_choice_box)
    self.select_song_thread.button_elements_signal.connect(self.update_button_elements)
    self.select_song_thread.clear_button_elements.connect(self.clear_button_elements)
    self.select_song_thread.clear_key_choice_box.connect(self.clear_key_choice_box)
    self.select_song_thread.started.connect(self.lock_inputs)
    self.select_song_thread.finished.connect(self.unlock_inputs)
    self.select_song_thread.start()
    print("Stopping select_song function")

@pyqtSlot()
def select_key(self):
    """Start the thread to select a key for the chosen song."""
    print("Starting select_key function")

    self.lock_inputs()

    selected_key = self.key_choice_box.currentText()

    self.select_key_thread = SelectKeyThread(self.driver, selected_key, self.button_elements)
    self.select_key_thread.log_updated.connect(self.update_log)
    self.select_key_thread.progress.connect(self.updateProgressBar)
    self.select_key_thread.status.connect(self.updateStatusLabel)
    self.select_key_thread.started.connect(self.lock_inputs)
    self.select_key_thread.finished.connect(self.unlock_inputs)
    self.select_key_thread.start()

    print("Stopping select_key function")

@pyqtSlot()
def download_and_process_images(self):
    """Start the thread to download and process images based on the selected song and key."""
    print("Starting download_and_process_images function")

    driver = self.driver
    key_choice_text = self.key_choice_box.currentText()
    selected_song_title = self.song_choice_box.currentText().split('\n')[0]
    selected_song_artist = self.song_choice_box.currentText().split('\n')[1]
    paths = self.paths

```

```

download_horn_only = self.horn_checkbox.isChecked()

self.download_and_process_images_thread = DownloadAndProcessThread(driver, key_choice_te
self.download_and_process_images_thread.log_updated.connect(self.update_log)
self.download_and_process_images_thread.progress.connect(self.updateProgressBar)
self.download_and_process_images_thread.status.connect(self.updateStatusLabel)
self.download_and_process_images_thread.started.connect(self.lock_inputs)
self.download_and_process_images_thread.finished.connect(self.unlock_inputs)
self.download_and_process_images_thread.start()
print("Stopping download_and_process_images function")

@pyqtSlot(list)
def update_button_elements(self, new_elements):
    """Update the list of button elements."""
    self.button_elements = new_elements

@pyqtSlot(str)
def update_song_info(self, new_song_info):
    """Update the song info list."""
    self.song_info.append(new_song_info)

@pyqtSlot(str)
def update_song_choice_box(self, new_choice):
    """Update the song choice box with a new choice."""
    self.song_choice_box.addItem(new_choice)

@pyqtSlot(str)
def update_key_choice_box(self, new_choice):
    """Update the key choice box with a new choice."""
    self.key_choice_box.addItem(new_choice)

@pyqtSlot(str)
def update_log(self, new_log):
    """Update the log area with a new log entry."""
    self.append_log(new_log)

@pyqtSlot(int)
def insert_separator_slot(self, index):
    """Insert a separator in the song choice box."""
    self.song_choice_box.insertSeparator(index)

@pyqtSlot()
def clear_song_info(self):
    """Clear the song info list."""
    self.song_info.clear()

@pyqtSlot()
def clear_song_choice_box(self):
    """Clear the song choice box."""
    self.song_choice_box.clear()

@pyqtSlot()
def clear_key_choice_box(self):
    """Clear the key choice box."""
    self.key_choice_box.clear()

@pyqtSlot()
def clear_button_elements(self):
    """Clear the list of button elements."""
    self.button_elements.clear()

@pyqtSlot()

```

```

def send_song_choice_box_count(self):
    """Send the count of items in the song choice box."""
    count = self.song_choice_box.count()
    self.find_songs_thread.receive_song_choice_box_count.emit(count)

@pyqtSlot()
def checkbox_state_changed(self):
    """Update the download_horn_only flag based on the checkbox state."""
    state = self.horn_checkbox.isChecked()
    self.download_and_process_images_thread.download_horn_only = state

```

Application Execution

```

In [9]: # Initialize the application
app = QApplication(sys.argv)

# Initialize our class
app_window = App()

# Show the window
app_window.show()

# Execute the application
sys.exit(app.exec_())

```

Creating GUI elements

Adding GUI elements to GUI

Closing driver

An exception has occurred, use %tb to see the full traceback.

SystemExit: 0

```

/home/sd205521/anaconda3/envs/rapids-23.12/lib/python3.10/site-packages/IPython/core/interactiveshell.py:3561: UserWarning: To exit: use 'exit', 'quit', or Ctrl-D.
warn("To exit: use 'exit', 'quit', or Ctrl-D.", stacklevel=1)

```