safe_json_dumps Function

```
def safe_json_dumps(data):
    """Safely serialize the data to JSON format, excluding non-serializable types."""
    try:
        # Attempt to convert the entire data dictionary to JSON with indentation of 4 spaces
        # This is the happy path where all data is JSON-serializable
        return json.dumps(data, indent=4)
    except TypeError:
        # This exception occurs when the data contains objects that can't be converted to JSON
        # For example, functions, complex objects, or file handles
        clean_data = {}
        # Iterate through each key-value pair in the original data dictionary
        for k, v in data.items():
            try:
                # Try to serialize just this single key-value pair to JSON
                # This tests if this specific item can be serialized
                json.dumps({k: v})
                # If the line above doesn't raise an exception, this item is serializable
                # So we add it to our clean_data dictionary
                clean_data[k] = v
            except TypeError:
                # If serializing this item fails, we convert it to a string representation
                # This ensures we retain some information about the value even if it's not JSON compatible
                clean_data[k] = str(v)
        # Finally, convert our cleaned dictionary to JSON and return it
        return json.dumps(clean_data, indent=4)
```

---

is_ffmpeg_installed Function

```
def is_ffmpeg_installed():
    """Checks if FFmpeg is installed by verifying if it's in the system path."""
    # shutil.which searches for the executable in the directories listed in PATH
    # It returns the path to the executable if found, or None if not found
    # This is a cross-platform way to check if a program is installed and available
    return shutil.which("ffmpeg") is not None
```

---

validate_directory Function

```
def validate_directory(directory):
    """Checks if the given directory exists and is writable."""
    # os.path.isdir checks if the path exists and is a directory (not a file)
    # os.access checks if the current process has specific permissions on the path
    # os.W_OK tests for write permission
    # Both conditions must be true for the function to return True
    return os.path.isdir(directory) and os.access(directory, os.W_OK)
```

---

is_valid_youtube_url Function

```
def is_valid_youtube_url(url):
    """Validates if a given URL is a valid YouTube link."""
    # Define a regular expression pattern that matches YouTube URLs
    # ^ anchors the match to the start of the string
    # (https?\:\/\/)? means the http:// or https:// part is optional
    # (www\.youtube\.com|youtu\.?be) matches either www.youtube.com or youtu.be or youtube.be
    # \/.+ requires at least one character after the slash
    # $ anchors the match to the end of the string
    youtube_regex = r'^(https?\:\/\/)?(www\.youtube\.com|youtu\.?be)\/.+$'
    # re.match checks if the pattern matches at the beginning of the string
    # Returns a match object if there's a match, or None if there's no match
    # This is converted to True or False
    return re.match(youtube_regex, url)
```

---

sanitize_filename Function

```
def sanitize_filename(filename):
    """Sanitizes a filename by removing invalid characters and normalizing."""
    # unicodedata.normalize converts unicode characters to a standard form
    # 'NFKD' means "Normalization Form Compatibility Decomposition"
    # This converts special characters like accented letters to their basic form
    filename = unicodedata.normalize('NFKD', filename)
    # Replace spaces with underscores to avoid issues with spaces in filenames
    filename = filename.replace(' ', '_')
    # Use a regular expression to remove any characters that aren't:
    # \w: word characters (letters, digits, underscore)
```

```
    # \-: hyphens
    # _: underscores
    # \.: periods
    # This ensures the filename only contains safe characters for all file systems
    filename = re.sub(r'[^\w\-_\.]', '', filename)
    # Return the sanitized filename
    return filename
```

---

progress_hook Function (Inside download_video_with_progress)

```
def progress_hook(d):
    # This is a callback function that gets called by yt-dlp with download progress information
    # d is a dictionary containing status information about the download

    # Check if the current status is 'downloading'
    if d['status'] == 'downloading':
        # Get the total size of the file in bytes
        # We try to get 'total_bytes' first, but if it's not available (None),
        # we fall back to 'total_bytes_estimate'
        total_bytes = d.get('total_bytes') or d.get('total_bytes_estimate')

        # Get the number of bytes downloaded so far
        # If 'downloaded_bytes' is not in the dictionary, default to 0
        downloaded_bytes = d.get('downloaded_bytes', 0)

        # Calculate the download progress as a percentage, but only if we have valid numbers
        if total_bytes and downloaded_bytes:
            # Calculate percentage and convert to integer (0-100)
            progress = int((downloaded_bytes / total_bytes) * 100)
        else:
            # If we can't calculate progress (maybe the total size is unknown), default to 0
            progress = 0

        # Call the progress_callback function with the calculated percentage
        # This will update the UI with the current progress
        progress_callback(progress)
```

```
    # Check if the status is 'finished', meaning the download part is complete
    elif d['status'] == 'finished':
        # Set progress to 95% instead of 100% because post-processing (like conversion) still needs to happen
        progress_callback(95)
```

---

download_video_with_progress Function

```
def download_video_with_progress(user_input, download_dir, output_format, progress_callback):
    """Downloads a YouTube video with progress tracking."""
    # progress_hook function is defined here (explained above)

    # Create a template for the output filename
    # It includes:
    # - The download directory
    # - The video title (limited to 100 characters)
    # - The selected output format
    # - The file extension (which will be set by yt-dlp)
    filename_template = os.path.join(download_dir, f'%(title).100s-{output_format}.%(ext)s')

    # Set up the basic options for yt-dlp
    ydl_opts = {
        'noplaylist': True,           # Don't download playlists, just the single video
        'progress_hooks': [progress_hook],  # Set the progress hook function we defined
        'outtmpl': filename_template,  # Set the output filename template
        'restrictfilenames': True,    # Restrict filenames to ASCII chars for compatibility
        'quiet': True,                # Don't print output to the console
        'no_warnings': True,          # Don't print warnings to the console
        'logger': None,               # Don't use a logger
    }

    # Try to find the FFmpeg executable in the system PATH
    ffmpeg_path = shutil.which("ffmpeg")
    # If FFmpeg is found, add its location to the options
    if ffmpeg_path:
        ydl_opts['ffmpeg_location'] = ffmpeg_path

    # Configure format-specific options based on the selected output format
```

```
if output_format == 'mp3':
    # Update the options for MP3 audio format
    ydl_opts.update({
        'format': 'bestaudio',  # Get the best audio quality
        'postprocessors': [{    # Set up post-processing for audio extraction
            'key': 'FFmpegExtractAudio',  # Use FFmpeg to extract audio
            'preferredcodec': 'mp3',      # Convert to MP3 format
            'preferredquality': '192',    # Set audio quality to 192kbps
        }],
    })
elif output_format == 'mp4':
    # Update the options for MP4 video format
    ydl_opts.update({
        'format': 'best[ext=mp4]/best',  # Try to get MP4, otherwise get best quality
        # Add FFmpeg parameters to help with codec detection
        'postprocessor_args': {
            'ffmpeg': ['-analyzeduration', '100M', '-probesize', '100M']
        },
    })
elif output_format == 'mov':
    # Update the options for MOV video format
    ydl_opts.update({
        'format': 'best[ext=mp4]/best',  # First get the best mp4/video format
        'merge_output_format': 'mov',    # Set the output format to MOV
    })

try:
    # Log the start of the download
    logging.info(f"Starting download for {output_format} format")

    # Create a YoutubeDL object with the configured options
    # The 'with' statement ensures proper cleanup when done
    with yt_dlp.YoutubeDL(ydl_opts) as ydl:
        # First extract information about the video without downloading it
        # This lets us get the title and other metadata
        info = ydl.extract_info(user_input, download=False)

        # Get the video title from the info dictionary
```

```
            # If 'title' is not in the dictionary, use 'Unknown title' as a fallback
            video_title = info.get('title', 'Unknown title')

            # Log the video title
            logging.info(f"Downloading: {video_title}")

            # Start the actual download
            # ydl.download takes a list of URLs to download
            ydl.download([user_input])

        # Log completion of download
        logging.info(f"Download completed: {video_title}")

        # Set progress to 100% when all processing is done
        progress_callback(100)

    except Exception as e:
        # If any error occurs during the download process
        # Log the error with detailed traceback
        logging.error(f"Download error: {e}", exc_info=True)

        # Re-raise the exception so it can be caught by the calling function
        # This allows the UI to display an error message
        Raise
```

---

change_directory Method (in DownloadManagerApp class)

```
def change_directory(self):
    """Change the download directory."""
    # Open a directory selection dialog
    # initialdir sets the starting directory to the current download directory
    # title sets the title of the dialog window
    new_dir = filedialog.askdirectory(initialdir=self.download_directory, title="Select Download Directory")

    # If a directory was selected (not canceled)
    # askdirectory returns an empty string if the user cancels
    if new_dir:
```

```
        # Clear the current directory entry field
        # 0 is the start position, tk.END is the end position
        self.dir_entry.delete(0, tk.END)

        # Insert the new directory path into the entry field
        self.dir_entry.insert(0, new_dir)

        # Update the download_directory instance variable
        self.download_directory = new_dir
```

---

update_progress Method (in DownloadManagerApp class)

```
def update_progress(self, value):
    """Updates the progress bar."""
    # Set the progress bar value (0-100)
    self.progress['value'] = value

    # Update status text based on progress value
    # If progress is between 95% and 100%, show "Converting..."
    # Otherwise, show the download percentage
    status_text = "Converting..." if value >= 95 and value < 100 else f"Downloading: {value}%"

    # Update the status label text
    self.status_label.config(text=status_text)

    # Force an update of the GUI
    # This ensures the progress bar and status label are updated immediately
    # without waiting for the next natural GUI update cycle
    self.root.update_idletasks()
```

---

download_thread_func Function (inside download_video method)

```
def download_thread_func():
    # This function runs in a separate thread to handle the download
    # This prevents the GUI from freezing during the download

    try:
```

```
        # Start the download with progress tracking
        # This calls the download_video_with_progress function we defined earlier
        # and passes the update_progress method as the progress_callback
        download_video_with_progress(user_input, download_dir, output_format, self.update_progress)

        # Schedule updating the status label on the main thread
        # root.after schedules a function to run after a delay (0ms means ASAP)
        # lambda: creates an anonymous function that calls status_label.config
        # This is necessary because you can't directly update GUI elements from a background thread
        self.root.after(0, lambda: self.status_label.config(text="Download completed!"))

        # Schedule showing a success message on the main thread
        self.root.after(0, lambda: messagebox.showinfo("Success", "The video has been downloaded successfully."))

    except Exception as e:
        # If an error occurs during download

        # Get the error message as a string
        error_msg = str(e)

        # Schedule updating the status label with the error on the main thread
        # If error message is too long, it's truncated to 50 characters
        self.root.after(0, lambda: self.status_label.config(
            text=f"Error: {error_msg[:50]}..." if len(error_msg) > 50 else f"Error: {error_msg}"
        ))

        # Schedule showing an error message on the main thread
        self.root.after(0, lambda: messagebox.showerror("Download Failed", f"An error occurred: {error_msg}"))

    finally:
        # This block always runs, whether the download succeeds or fails

        # Schedule re-enabling the download button on the main thread
        self.root.after(0, lambda: self.download_button.config(state=tk.NORMAL))
```

---

toggle_dark_mode Method (in DownloadManagerApp class)

```
def toggle_dark_mode(self):
    """Toggles dark mode on and off."""
    # Determine colors based on current mode
    # If not in dark mode, use dark colors; otherwise, use light colors

    # Background color for most widgets
    bg_color = "#2E2E2E" if not self.dark_mode else "white"

    # Foreground (text) color for most widgets
    fg_color = "white" if not self.dark_mode else "black"

    # Background color for entry fields
    entry_bg = "#3E3E3E" if not self.dark_mode else "white"

    # Text color for entry fields
    entry_fg = "white" if not self.dark_mode else "black"

    # Configure the root window background
    self.root.configure(bg=bg_color)

    # Update all widgets with new colors
    # root.winfo_children() returns a list of all widgets in the root window
    for widget in self.root.winfo_children():
        # Get the widget's class name (type)
        widget_type = widget.winfo_class()

        # Check if it's a widget type that needs color update
        if widget_type in ("Label", "Button", "Frame", "Radiobutton"):
            try:
                # Try to configure background and foreground colors
                widget.configure(bg=bg_color, fg=fg_color)
            except tk.TclError:
                # Some widgets might not support these options
                # If configuration fails, just skip it
                pass

        elif widget_type == "Entry":
            try:
```

```
                # Configure entry fields with their specific colors
                widget.configure(bg=entry_bg, fg=entry_fg)
            except tk.TclError:
                # Skip if configuration fails
                pass

    # Toggle the dark mode flag
    # This flips the value from True to False or from False to True
    self.dark_mode = not self.dark_mode
```

The lambda Functions

Let me explain the lambda functions in more detail:

1. lambda: self.status_label.config(text="Download completed!")
    - This creates an anonymous function that takes no arguments
    - When called, it updates the status label text to "Download completed!"
    - It's used with root.after() to schedule this update on the main thread
2. lambda: self.status_label.config(text=f"Error: {error_msg[:50]}..." if len(error_msg) > 50 else f"Error: {error_msg}")
    - This creates an anonymous function that takes no arguments
    - It contains a conditional expression that:
        - If the error message is longer than 50 characters, shows the first 50 characters followed by "..."
        - Otherwise, shows the full error message
    - The function updates the status label with this text when called
3. lambda: self.download_button.config(state=tk.NORMAL)
    - This creates an anonymous function that takes no arguments
    - When called, it enables the download button by setting its state to NORMAL
    - It's used in the finally block to ensure the button is re-enabled whether the download succeeds or fails

These lambda functions are particularly useful when you need to pass a function as an argument (like to root.after()) but don't want to define a separate named function for a simple operation.