Folder: plant

File: finalclassifierwith3attribute.py

Documented code for finalclassifierwith3attribute.py:

```
# Import necessary libraries
import time
import numpy as np
import pandas as pd
import tensorflow as tf
import keras
from keras.preprocessing.image import ImageDataGenerator
from keras import applications
from keras.models import Sequential, load_model
from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D, Flatten, Dense, Dropout
from keras.preprocessing import image
import cv2
import warnings
warnings.filterwarnings('ignore')
import os

# Mount Google Drive
drive.mount("/content/drive")

# Load the dataset
labels = pd.read_csv("/content/drive/MyDrive/PhD Work/InceptionResNetV2 architecture 3Attribute/KaranjNeemDataset (1).csv")
labels = labels.sample(frac=1)
labels["id"] = labels["image"]

# Data augmentation and pre-processing
gen = ImageDataGenerator(
    rescale=1./255.,
```

```
    horizontal_flip = True,
    validation_split=0.2
)

train_generator = gen.flow_from_dataframe(
    labels,
                directory = "/content/drive/MyDrive/PhD Work/InceptionResNetV2 architecture
3Attribute/Allinone3Attribute",
    x_col = 'image',
    y_col = 'type',
    subset="training",
    color_mode="rgb",
    target_size = (331,331),
    class_mode="categorical",
    batch_size=32,
    shuffle=True,
    seed=42,
)

validation_generator = gen.flow_from_dataframe(
    labels,
                directory = "/content/drive/MyDrive/PhD Work/InceptionResNetV2 architecture
3Attribute/Allinone3Attribute",
    x_col = 'image',
    y_col = 'type',
    subset="validation",
    color_mode="rgb",
    target_size = (331,331),
    class_mode="categorical",
    batch_size=32,
    shuffle=True,
    seed=42,
)
```

```
# Load the InceptionResNetV2 architecture with imagenet weights as base
base_model = tf.keras.applications.InceptionResNetV2(
    include_top=False,
    weights='imagenet',
    input_shape=(331,331,3)
)

# Freeze the base model layers
base_model.trainable=False

# Build the model
model = tf.keras.Sequential([
    base_model,
    tf.keras.layers.BatchNormalization(renorm=True),
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(6, activation='softmax')
])

# Compile the model
model.compile(optimizer='Adam',loss='categorical_crossentropy',metrics=['accuracy'])

# Train the model
early = tf.keras.callbacks.EarlyStopping(patience=10,
                                         min_delta=0.001,
                                         restore_best_weights=True)

STEP_SIZE_TRAIN = train_generator.n//train_generator.batch_size
STEP_SIZE_VALID = validation_generator.n//validation_generator.batch_size
```

```
history = model.fit(train_generator,
                    steps_per_epoch=STEP_SIZE_TRAIN,
                    validation_data=validation_generator,
                    validation_steps=STEP_SIZE_VALID,
                    epochs=5,
                    callbacks=[early])

# Save the model
model.save("/content/drive/MyDrive/PhD Work/InceptionResNetV2 architecture/3Attribute3AttributePlantModel.h5")

# Evaluate the model
accuracy_score = model.evaluate(validation_generator)
print(accuracy_score)
print("Accuracy: {:.4f}%".format(accuracy_score[1] * 100))

print("Loss: ",accuracy_score[0])

# Test the model
test_img_path = "/content/drive/MyDrive/PhD Work/InceptionResNetV2 architecture 3Attribute/Allinone3Attribute/1456.jpg"
img = cv2.imread(test_img_path)
resized_img = cv2.resize(img, (331, 331)).reshape(-1, 331, 331, 3)/255
prediction = model.predict(resized_img)
print(class_names[np.argmax(prediction)])

# Print medicinal properties of the plant detected
class_prediction=class_names[np.argmax(prediction)]

if class_prediction == 'Karanj Trunk' or class_prediction == 'Karanj Leaf' or class_prediction == 'Karanj Seed':
    print('Karanj Popularly known as Indian Beech in outside India is a medicinal herb used mainly for
```

skin disorders. Karanja oil is applied to the skin to manage boils, rashes, and eczema as well as heal wounds due to its antimicrobial properties. The oil can also be useful in arthritis due to its anti-inflammatory activities.')

if class_prediction == 'Neem Trunk' or class_prediction == 'Neem Leaf' or class_prediction == 'Neem Seed':
    print('Neem is a versatile medicinal tree. Neem oil and neem leaves are used for various medicinal purposes. It has anti-inflammatory, antifungal, and antibacterial properties, making it beneficial for skin care, hair care, and managing various health conditions.')

if class_prediction == 'Peeple Trunk' or class_prediction == 'Peeple Leaf' or class_prediction == 'Peeple Seed':
    print('Peepal: The bark of the Peeple tree, rich in vitamin K, is an effective complexion corrector and preserver. It also helps in various ailments such as Strengthening blood capillaries, minimising inflammation, Healing skin bruises faster, increasing skin resilience, treating pigmentation issues, wrinkles, dark circles, lightening surgery marks, scars, and stretch marks.')

# Calculate the elapsed time
end_time = time.time()
elapsed_time = end_time - start_time
print(f"Time taken: {elapsed_time:.2f} seconds")
Time_in_Minute = elapsed_time / 60
print(f"Time taken: {Time_in_Minute:.2f} minutes")
```

**Explanation:**

1. **Line 1-5: Import Statements**
    - We import necessary libraries such as `time`, `numpy`, `pandas`, `tensorflow`, `keras`, `ImageDataGenerator`, `applications`, `Sequential`, `load_model`, `Conv2D`, `MaxPooling2D`, `GlobalAveragePooling2D`, `Flatten`, `Dense`, `Dropout`, `image`, `cv2`, `warnings`, and `os`.

2. **Line 6-10: User Input Handling**
```

- We mount Google Drive to access the dataset.

3. **Line 11-15: Variable Declarations**
   - We load the dataset and perform some pre-processing.

4. **Line 16-20: Data Augmentation and Pre-processing**
   - We create an `ImageDataGenerator` object for data augmentation and pre-processing.

5. **Line 21-25: Control Flow - Part 1**
   - We split the dataset into training and validation sets.

6. **Line 26-30: Function Calls - Part 1**
   - We load the InceptionResNetV2 architecture with imagenet weights as base.

7. **Line 31-35: Control Flow - Part 2**
   - We freeze the base model layers.

8. **Line 36-40: Function Calls - Part 2**
   - We build the model by adding additional layers on top of the base model.

9. **Line 41-45: Output Generation**
   - We compile the model with the specified optimizer, loss function, and metrics.

10. **Line 46-50: Error Handling**
    - We define an early stopping callback to prevent overfitting.

11. **Line 51-55: Looping Structures**
    - We train the model using the training and validation data.

12. **Line 56-60: Data Manipulation**
    - We save the trained model.

13. **Line 61-65: Control Flow - Part 3**

- We evaluate the model on the validation data.

14. **Line 66-70: Function Calls - Part 3**
    - We load an image, pre-process it, and make a prediction.

15. **Line 71-75: Output Generation**
    - We print the predicted class and the medicinal properties of the plant detected.

16. **

Folder: app

Folder: media

Folder: plant

Folder: templates

File: manage.py

Documented code for manage.py:

Code Explanation:

```
#!/usr/bin/env python
"""Django's command-line utility for administrative tasks."""
```

1. **Line 1: Shebang (Optional)**
    - This line is a shebang, indicating that the script should be executed using the Python interpreter.
    - It's optional and typically used when the script is intended to be executed directly as a command.

```
import os
import sys
```

2. **Line 2-3: Import Statements**

- These lines import the `os` and `sys` modules.
- The `os` module provides functions for interacting with the operating system.
- The `sys` module provides access to system-specific parameters and functions.

```
def main():
    """Run administrative tasks."""
    os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'plant.settings')
```

3. **Line 6-8: Main Function**
   - The `main()` function is the entry point of the script.
   - It sets the `DJANGO_SETTINGS_MODULE` environment variable to 'plant.settings'.
   - This environment variable is used by Django to locate the project's settings module.

```
    try:
        from django.core.management import execute_from_command_line
    except ImportError as exc:
        raise ImportError(
            "Couldn't import Django. Are you sure it's installed and "
            "available on your PYTHONPATH environment variable? Did you "
            "forget to activate a virtual environment?"
        ) from exc
```

4. **Line 10-15: Error Handling**
   - This `try-except` block attempts to import the `execute_from_command_line` function from Django's `django.core.management` module.
   - If the import fails, it raises an `ImportError` with a custom error message.
   - The custom error message provides guidance on potential issues like missing Django installation or an inactive virtual environment.

```
    execute_from_command_line(sys.argv)
```

5. **Line 17: Function Call**
   - This line calls the `execute_from_command_line()` function from the `django.core.management` module.
   - It passes the command-line arguments (stored in `sys.argv`) as an argument to this function.

```
if __name__ == '__main__':
    main()
```

6. **Line 20-21: Conditional Execution**
   - This conditional statement checks if the script is being run as the main program (i.e., not imported as a module).
   - If it is the main program, it calls the `main()` function.

Conclusion:

The `manage.py` script is a command-line utility for Django projects. It allows users to perform various administrative tasks, such as creating migrations, running the development server, and creating superusers. The script imports necessary modules, sets environment variables, handles errors, and calls the `execute_from_command_line()` function to execute Django management commands.

File: result.html
Documented code for result.html:
Explanation of result.html:

1. Import Statements:

```html
<!DOCTYPE html>
```

This line specifies that the document follows the HTML5 standard. It is a declaration that tells the browser to interpret the document as HTML5.

2. HTML Structure:

```html
<html>
<head>
    <title>Recognition Result</title>
    <style>
        ...
    </style>
</head>
<body>
    ...
</body>
</html>
```

This code defines the basic structure of an HTML document. It includes the `<html>`, `<head>`, and `<body>` tags, which are essential for creating a web page.

3. Styling:

```html
<style>
    ...
```

</style>
```

This section contains CSS styles that define the appearance of the web page. It includes rules for the `body`, `.result-box`, `img`, and `p` elements. These styles control the layout, colors, fonts, and other visual aspects of the page.

**4. Result Box:**

```html
<div class="result-box">
    ...
</div>
```

This div element with the class "result-box" serves as a container for the recognition result. It has a fixed maximum width, padding, border-radius, box-shadow, and a white background color. It is centered both horizontally and vertically within the page.

5. Heading:

```html
<h2>Recognition Result:</h2>
```

This heading element displays the text "Recognition Result:" within the result box. It is used to introduce the purpose of the page.

6. Uploaded Image:

```html
<img src="{{ uploaded_image }}" alt="Uploaded Image">
```
```

This image element displays the uploaded image. The `src` attribute is a placeholder for the actual image source, which is likely generated dynamically based on user input or a server-side process. The `alt` attribute provides alternative text for the image, which is useful for accessibility purposes.

7. Class Prediction:

```html
<p>Class Prediction: {{ result.0 }}</p>
```

This paragraph element displays the predicted class for the uploaded image. The `{{ result.0 }}` placeholder will be replaced with the actual class prediction, which is likely obtained from a machine learning model or a server-side process.

8. Additional Information:

```html
<p>{{ result.1 }}</p>
```

This paragraph element can be used to display additional information related to the recognition result. The `{{ result.1 }}` placeholder will be replaced with the actual information, which could be a confidence score, a description of the predicted class, or any other relevant data.

Conclusion:

This HTML template defines the structure and styling for a web page that displays the result of an image recognition process. It includes a result box, a heading, an image display area, and placeholders for the class prediction and additional information. The actual values for these placeholders are likely generated dynamically based on user input or a server-side process.

File: upload.html

Documented code for upload.html:

Code Explanation:

1. Import Statements (Line 1-5):

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Image Upload</title>
    <!-- Bootstrap CSS CDN -->
<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css" integrity="sha384-B4gt1jrGC7Jh4AgTPSdUtOBvfO8sh+WytkY3Y5CcH25PLOiMZ95ES9B3xUZUn6N" crossorigin="anonymous">
```

- This section includes the necessary HTML tags and meta information for the web page.
- It also imports the Bootstrap CSS CDN, which provides styling for the page.

2. HTML Structure and Styling (Line 6-20):

```html
<style>
    body {
        background-color: #f8f9fa;
        color: #343a40;
        height: 100vh;
        display: flex;
        align-items: center;
        justify-content: center;
```

```
        margin: 0;
    }
    .container {
        max-width: 400px;
        padding: 20px;
        border-radius: 8px;
        box-shadow: 0 4px 6px rgba(0, 0, 0, 0.1);
        background-color: #fff;
    }
    h2 {
        text-align: center;
        margin-bottom: 30px;
    }
    form {
        padding: 20px;
    }
    button {
        background-color: #007bff;
        color: #fff;
    }
</style>
```

- This section contains CSS styling for the web page.
- It defines styles for the body, container, heading, form, and button elements.

3. HTML Form (Line 21-30):

```html
<body>
    <div class="container">
        <h2>Upload Image</h2>
        <form method="post" enctype="multipart/form-data">
```

```
        {% csrf_token %}
        {{ form }}
        <button type="submit" class="btn btn-primary btn-block">Upload Image</button>
    </form>
</div>
```

- This section creates an HTML form for uploading an image.
- It includes a heading, a form with a CSRF token and the form fields generated by Django, and a submit button.

4. Bootstrap JS and Popper.js CDN (Line 31-35):

```html
<!-- Bootstrap JS and Popper.js CDN (required for Bootstrap) -->
<script src="https://code.jquery.com/jquery-3.5.1.slim.min.js" integrity="sha384-DfXdz2htPH0lsSSs5nCTpuj/zy4C+OGpamoFVy38MVBnE+IbbVYUew+OrCXaRkfj" crossorigin="anonymous"></script>
<script src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.11.6/dist/umd/popper.min.js" integrity="sha384-dQVlziZ8DEghGHiP99R8Tz9DJks7Cd7P3MZweelF+0GnyDwr6bVEwH+WrGzprY" crossorigin="anonymous"></script>
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js" integrity="sha384-B4gt1jrGC7Jh4AgTPSdUtOBvfO8sh+WytkY3Y5CcH25PLOiMZ95ES9B3xUZUn6N" crossorigin="anonymous"></script>
</body>
</html>
```

- This section includes the necessary JavaScript libraries for Bootstrap to function properly.
- It includes jQuery, Popper.js, and Bootstrap JS.

Conclusion:

This HTML code creates a web page with a form for uploading an image. It uses Bootstrap for styling and includes the necessary JavaScript libraries for Bootstrap to work. When a user selects an image and clicks the "Upload Image" button, the form data, including the image, is submitted to the server for processing.

Folder: __pycache__
File: asgi.py
Documented code for asgi.py:
Code Explanation:

Line 1: Import Statements

```
import os
```

This line imports the `os` module, which provides a portable way to use operating system-dependent functionality. It is used later in the code to set the Django settings module.

Line 2: Importing Django ASGI Application

```
from django.core.asgi import get_asgi_application
```

This line imports the `get_asgi_application` function from the `django.core.asgi` module. This function is used to create an ASGI application for a Django project.

Line 3: Setting Django Settings Module

```
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'plant.settings')
```

This line sets the `DJANGO_SETTINGS_MODULE` environment variable to the value `'plant.settings'`. This tells Django which settings module to use when starting up. The `plant.settings` module contains the Django settings for the `plant` project.

Line 4: Creating ASGI Application

```
application = get_asgi_application()
```

This line calls the `get_asgi_application()` function to create an ASGI application for the Django project. The ASGI application is an entry point for ASGI-compatible web servers, such as Uvicorn or Gunicorn.

Conclusion:

The `asgi.py` file in a Django project serves as the entry point for ASGI-compatible web servers. It imports the necessary modules, sets the Django settings module, and creates an ASGI application. This allows Django to be deployed and served by ASGI-based web servers.

File: wsgi.py
Documented code for wsgi.py:
Code Explanation:

Line 1: Import Statements

```
import os
```

This line imports the `os` module, which provides a portable way to use operating system-dependent functionality. It is used later in the code to set the Django settings module.

Line 2: WSGI Application Import

```
from django.core.wsgi import get_wsgi_application
```

This line imports the `get_wsgi_application` function from the `django.core.wsgi` module. This function is used to create a WSGI application object that can be used to serve Django requests.

Line 3: Setting Django Settings Module

```
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'plant.settings')
```

This line sets the `DJANGO_SETTINGS_MODULE` environment variable to the string `'plant.settings'`. This environment variable is used by Django to determine which settings module to use. In this case, the settings module is located in the `plant` package and is named `settings.py`.

Line 4: Creating WSGI Application Object

```
application = get_wsgi_application()
```

This line calls the `get_wsgi_application()` function to create a WSGI application object. This object is used to serve Django requests.

Conclusion:

The `wsgi.py` file is a WSGI configuration file for a Django project. It imports the necessary modules, sets the Django settings module, and creates a WSGI application object. This file is used by the web

server to serve Django requests.

File: settings.py
Documented code for settings.py:
Code Explanation:

1. Import Statements:

```
from pathlib import Path
```

- This line imports the `Path` class from the `pathlib` module. The `Path` class is used to manipulate file and directory paths in a portable way.

2. Variable Declarations:

```
BASE_DIR = Path(__file__).resolve().parent.parent
```

- This line defines a variable called `BASE_DIR` and sets its value to the absolute path of the project directory. It uses the `Path` class to resolve the path and then traverses up two levels to reach the project root.

3. User Input Handling:

- This section is not applicable as there is no user input handling in the provided code snippet.

4. Control Flow - Part 1:

- This section is also not applicable as there are no conditional statements or control flow structures in the provided code snippet.

5. Function Calls - Part 1:

- This section is not applicable as there are no function calls in the provided code snippet.

6. Looping Structures:

- This section is not applicable as there are no looping structures in the provided code snippet.

7. Data Manipulation:

- This section is not applicable as there are no data manipulation operations in the provided code snippet.

8. Control Flow - Part 2:

- This section is not applicable as there are no additional conditional statements or control flow structures in the provided code snippet.

9. Function Calls - Part 2:

- This section is not applicable as there are no additional function calls in the provided code snippet.

10. Output Generation:

- This section is not applicable as there is no output generation in the provided code snippet.

11. Error Handling:

- This section is not applicable as there are no error handling mechanisms in the provided code snippet.

Conclusion:

The provided code snippet contains basic Django project settings and configurations. It includes settings for the project's base directory, database, security, and other general Django settings. This code is typically found in the `settings.py` file of a Django project and is used to configure the project's behavior and environment.

File: urls.py
Documented code for urls.py:
Code Explanation:

1. Import Statements:

```
from django.contrib import admin
from django.urls import path, include
from django.conf import settings
from django.conf.urls.static import static
```

- `from django.contrib import admin`: Imports the Django admin module, which provides a user interface for managing data in the database.
- `from django.urls import path, include`: Imports the `path()` and `include()` functions from Django's URL routing module.
- `from django.conf import settings`: Imports Django's settings module, which contains various configuration options for the project.
- `from django.conf.urls.static import static`: Imports the `static()` function for serving static files during development.

2. Variable Declarations:

```
urlpatterns = []
```

- `urlpatterns`: A list that will contain URL patterns for the project. Each URL pattern defines a mapping between a URL and a view function or class-based view.

3. User Input Handling:

There is no user input handling in this code snippet.

4. Control Flow - Part 1:

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('app.urls')),
]
```

- `path('admin/', admin.site.urls)`: Defines a URL pattern for the Django admin interface. When a user visits the `/admin/` URL, they will be directed to the admin site.
- `path('', include('app.urls'))`: Defines a URL pattern for the `app` application. When a user visits the root URL (`/`), they will be directed to the `app` application's URL configuration, defined in the `app/urls.py` file.

5. Function Calls - Part 1:

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('app.urls')),
]
```

- `admin.site.urls`: This is a function that returns the URL patterns for the Django admin interface.

- `include('app.urls')`: This is a function that includes the URL patterns defined in the `app/urls.py` file.

6. Looping Structures:

There are no looping structures in this code snippet.

7. Data Manipulation:

There is no data manipulation in this code snippet.

8. Control Flow - Part 2:

```
if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

- `if settings.DEBUG:`: Checks if the `DEBUG` setting in Django's settings module is set to `True`. This setting is typically set to `True` during development and `False` in production.

9. Function Calls - Part 2:

```
if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

- `static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)`: This function serves static files during development. It takes two arguments:
  - `settings.MEDIA_URL`: The URL prefix for static files.
  - `settings.MEDIA_ROOT`: The directory where static files are stored.

10. Output Generation:

There is no output generation in this code snippet.

11. Error Handling:

There is no error handling in this code snippet.

Conclusion:

This code snippet defines the URL patterns for a Django project. It includes URL patterns for the Django admin interface and the `app` application. It also serves static files during development if the `DEBUG` setting is set to `True`.

Folder: __pycache__
Folder: migrations
File: admin.py
Documented code for admin.py:
1. **Line 1: Import Statements**

```
from django.contrib import admin
```

This line imports the `admin` module from the `django.contrib` package. The `admin` module provides functionality for creating an administrative interface for Django models.

2. **Line 2-5: Variable Declarations**

There are no variable declarations in this code snippet.

3. **Line 6-10: User Input Handling**

There is no user input handling in this code snippet.

4. **Line 11-15: Control Flow - Part 1**

There is no control flow in this code snippet.

5. **Line 16-20: Function Calls - Part 1**

There are no function calls in this code snippet.

6. **Line 21-25: Looping Structures**

There are no looping structures in this code snippet.

7. **Line 26-30: Data Manipulation**

There is no data manipulation in this code snippet.

8. **Line 31-35: Control Flow - Part 2**

There is no control flow in this code snippet.

9. **Line 36-40: Function Calls - Part 2**

There are no function calls in this code snippet.

10. **Line 41-45: Output Generation**

There is no output generation in this code snippet.

11. **Line 46-50: Error Handling**

There is no error handling in this code snippet.

Conclusion:

The provided code snippet is a Django admin module configuration file. It does not contain any executable code and is used to register Django models with the admin interface. This allows administrators to manage the data in the database through a web-based interface.

File: apps.py
Documented code for apps.py:

```
from django.apps import AppConfig
```

This line imports the `AppConfig` class from the `django.apps` module. In Django, every application is represented by a Python package, and each package has an `AppConfig` class that provides metadata about the application.

```
class AppConfig(AppConfig):
```

This line defines a class named `AppConfig` that inherits from the `AppConfig` class imported from Django. This class will serve as the configuration class for our Django application.

```
    default_auto_field = 'django.db.models.BigAutoField'
```

This line sets the default auto-field for the application's models. The `default_auto_field` attribute specifies the default primary key field type for models in the application. In this case, it is set to `'django.db.models.BigAutoField'`, which is a 64-bit integer field that automatically increments.

```
```

```
    name = 'app'
```

This line sets the name of the application. The `name` attribute is used to identify the application within Django. It is typically set to the name of the Python package that contains the application.

In summary, this code defines a Django application configuration class named `AppConfig`. It sets the default auto-field for models in the application and specifies the application's name. This configuration class is essential for Django to recognize and manage the application.

File: urls.py
Documented code for urls.py:
Explanation:

1. Import Statements:

```
from django.urls import path
from .views import upload_image
```

- We import the `path` function from `django.urls` to define URL patterns.
- We also import the `upload_image` view from the current app's `views.py` file.

2. Variable Declarations:

```
app_name = 'your_app'
```

- We define a variable `app_name` and set it to 'your_app'. This is used to namespace the URL patterns for this app.

3. URL Patterns:

```
urlpatterns = [
    path('upload/', upload_image, name='upload_image'),
    # Add other URL patterns as needed
]
```

- We define a list called `urlpatterns` which contains URL patterns for this Django app.
- The first pattern matches the URL 'upload/' and associates it with the `upload_image` view.
- The `name` argument specifies a unique name for this URL pattern, which can be used in templates and other parts of the code to refer to this specific URL.

Conclusion:

The `urls.py` file in a Django app defines the URL patterns for that app. It maps incoming URLs to specific views, which handle the requests and generate responses. This allows us to create a clean and organized way to handle different types of requests in our Django application.

File: forms.py
Documented code for forms.py:
Code Explanation:

Line 1:

```
from django import forms
```

This line imports the `forms` module from the Django framework. Django is a popular Python web framework that provides a set of tools and libraries for building web applications. The `forms` module is used to create HTML forms and handle user input in Django applications.

Line 2:

```
from .models import UploadedImage
```

This line imports the `UploadedImage` model from the current Django application. Models in Django represent the data structures that are stored in the database. The `UploadedImage` model likely contains fields and methods for storing and managing uploaded images.

Line 4:

```
class ImageUploadForm(forms.ModelForm):
```

This line defines a new Django form class named `ImageUploadForm`. Django forms are used to handle user input and validate data before it is saved to the database. The `forms.ModelForm` class is a special type of form that is linked to a Django model. This means that the `ImageUploadForm` class will be able to interact with the `UploadedImage` model.

Line 5:

```
class Meta:
```

This line starts the definition of a nested `Meta` class within the `ImageUploadForm` class. The `Meta` class is used to configure the `ImageUploadForm` class and specify its properties.

Line 6:

```
model = UploadedImage
```

This line specifies the Django model that the `ImageUploadForm` class is associated with. In this case, the model is `UploadedImage`. This means that the `ImageUploadForm` class will be able to create and update instances of the `UploadedImage` model.

Line 7:

```
fields = ['image']
```

This line specifies the fields from the `UploadedImage` model that will be included in the `ImageUploadForm` class. In this case, the only field included is `image`. This means that the `ImageUploadForm` class will only have one field, which will be used to upload an image.

Conclusion:

The provided code defines a Django form class named `ImageUploadForm` that is linked to the `UploadedImage` model. This form class can be used to create and update instances of the `UploadedImage` model. The form class has one field, named `image`, which is used to upload an image.

Folder: __pycache__
File: 0001_initial.py
Documented code for 0001_initial.py:
Code Explanation:

1. Import Statements (Line 1):

```
```

```
from django.db import migrations, models
```

- This line imports the necessary modules from the Django framework.
- `migrations` module provides classes and functions for creating and managing database migrations.
- `models` module contains classes for defining database models and fields.

2. Class Definition (Line 4-45):

```
class Migration(migrations.Migration):

    initial = True

    dependencies = [
    ]

    operations = [
        migrations.CreateModel(
            name='UploadedImage',
            fields=[
                        ('id', models.BigAutoField(auto_created=True, primary_key=True, serialize=False, verbose_name='ID')),
                ('image', models.ImageField(upload_to='uploaded_images/')),
            ],
        ),
    ]
```

- This code defines a Django migration class named `Migration`.
- `initial` attribute is set to `True`, indicating that this is the initial migration for the app.
- `dependencies` attribute is an empty list, as this migration does not depend on any other

migrations.
- `operations` attribute is a list of operations to be performed during the migration.

3. Model Definition (Line 11-18):

```
migrations.CreateModel(
    name='UploadedImage',
    fields=[
                    ('id',  models.BigAutoField(auto_created=True,  primary_key=True,  serialize=False,
verbose_name='ID')),
        ('image', models.ImageField(upload_to='uploaded_images/')),
    ],
)
```

- This code defines a Django model named `UploadedImage`.
- It has two fields:
  - `id`: A primary key field that is automatically created and managed by Django.
    - `image`: An `ImageField` field that stores the uploaded image. The `upload_to` parameter specifies the directory where the images will be saved.

Conclusion:

This migration script defines the initial state of a Django app. It creates a model named `UploadedImage` with two fields: `id` and `image`. The `image` field is an `ImageField` that stores uploaded images in the `uploaded_images/` directory. This migration is necessary for the app to function properly and manage uploaded images.

File: views.py
Documented code for views.py:
Code Explanation:

1. Import Statements (Line 1-3):

- `from django.shortcuts import render`: Imports the `render` function from the Django shortcuts module. This function is used to render HTML templates in Django views.

- `from .forms import ImageUploadForm`: Imports the `ImageUploadForm` class from the `forms.py` module within the current Django app. This form class is used to handle image uploads.

- `from .ml_model import plant_recognition_model`: Imports the `plant_recognition_model` function from the `ml_model.py` module within the current Django app. This function performs plant recognition using a machine learning model.

2. Function Definition (Line 4):

- `def upload_image(request):`: Defines a Django view function named `upload_image`. This function handles HTTP requests for uploading and processing images.

3. Conditional Statement (Line 5-45):

- This conditional statement checks the HTTP request method. If the request method is `POST`, it means the user has submitted a form. Otherwise, it's assumed to be a `GET` request, indicating the initial page load.

4. Handling POST Requests (Line 6-40):

- `form = ImageUploadForm(request.POST, request.FILES)`: Creates an instance of the `ImageUploadForm` class with data from the HTTP request. This allows Django to validate the uploaded image.

- `if form.is_valid()`: Checks if the form data is valid. If it is, the following code is executed.

- `uploaded_image = form.save(commit=False)`: Saves the form data, but doesn't commit it to the database yet. This allows us to access the uploaded image file.

- `result = plant_recognition_model(uploaded_image.image)`: Calls the `plant_recognition_model` function with the uploaded image file as an argument. This function performs plant recognition using a machine learning model.

- `return render(request, 'result.html', {'result': result, 'uploaded_image': uploaded_image.image.url})`: Renders the `result.html` template with the plant recognition result and the URL of the uploaded image.

5. Handling GET Requests (Line 41-45):

- `form = ImageUploadForm()`: Creates an instance of the `ImageUploadForm` class without any data. This is used to display the image upload form on the initial page load.
- `return render(request, 'upload.html', {'form': form})`: Renders the `upload.html` template with the empty form.

Conclusion:

This Django view function, `upload_image`, handles both GET and POST requests. When a user submits an image upload form, it validates the form, saves the uploaded image, performs plant recognition using a machine learning model, and renders a result page with the recognition result and the uploaded image. When the page is initially loaded, it displays an empty image upload form.

File: models.py
Documented code for models.py:
Code Explanation:

Line 1: Import Statements

```
from django.db import models
```

This line imports the `models` module from the `django.db` package. The `models` module provides classes and functions for creating database models in Django, a popular Python web framework.

Line 2-5: Variable Declarations

```
class UploadedImage(models.Model):
    image = models.ImageField(upload_to='uploaded_images/')
```

In these lines, we define a Django model named `UploadedImage`. A model in Django represents a database table, and each instance of the model corresponds to a row in the table.

The `image` field is an `ImageField`, which is a Django field type specifically designed for storing images. The `upload_to` parameter specifies the directory where uploaded images will be stored relative to the Django project's media directory.

Line 6-10: User Input Handling

(Assuming there is no user input handling in this code snippet)

Line 11-15: Control Flow - Part 1

(Assuming there is no control flow in this code snippet)

Line 16-20: Function Calls - Part 1

(Assuming there are no function calls in this code snippet)

Line 21-25: Looping Structures

(Assuming there are no looping structures in this code snippet)

Line 26-30: Data Manipulation

(Assuming there is no data manipulation in this code snippet)

Line 31-35: Control Flow - Part 2

(Assuming there is no control flow in this code snippet)

Line 36-40: Function Calls - Part 2

(Assuming there are no function calls in this code snippet)

Line 41-45: Output Generation

(Assuming there is no output generation in this code snippet)

Line 46-50: Error Handling

(Assuming there is no error handling in this code snippet)

Conclusion:

The provided code snippet defines a Django model named `UploadedImage`. This model represents a database table for storing uploaded images. The `image` field is an `ImageField`, which is a Django field type specifically designed for storing images. The `upload_to` parameter specifies the directory where uploaded images will be stored relative to the Django project's media directory.

This code snippet is a fundamental part of a Django application that allows users to upload and store images. It provides the necessary database structure and field definitions for handling image uploads.

File: tests.py
Documented code for tests.py:
Code Explanation:

1. Import Statements (Line 1):

```
from django.test import TestCase
```

- This line imports the `TestCase` class from the `django.test` module.
- `TestCase` is a base class for writing test cases in Django, a popular Python web framework.

2. Variable Declarations (Lines 2-5):

```
class MyTestCase(TestCase):
    def setUp(self):
        self.assertEqual(1, 1)
```

- We define a test case class named `MyTestCase` that inherits from `TestCase`.
- Inside the `setUp` method, we use the `assertEqual` method to assert that `1` is equal to `1`.
- This is a simple assertion to ensure that the testing framework is working correctly.

3. User Input Handling (Lines 6-10):

```
    def test_something(self):
        user_input = input("Enter something: ")
        self.assertIsNotNone(user_input)
```

- We define a test method named `test_something`.
- Inside this method, we prompt the user to enter something using `input`.
- We then use the `assertIsNotNone` method to assert that the user input is not `None`.
- This ensures that the user has entered something.

4. Control Flow - Part 1 (Lines 11-15):

```
        if user_input == "hello":
            self.assertEqual(user_input, "hello")
```

- We use an `if` statement to check if the user input is equal to `"hello"`.
- If it is, we use the `assertEqual` method to assert that the user input is equal to `"hello"`.
- This ensures that the user has entered the correct value.

5. Function Calls - Part 1 (Lines 16-20):

```
    def test_function(self):
        result = my_function(1, 2)
        self.assertEqual(result, 3)
```

- We define another test method named `test_function`.
- Inside this method, we call the `my_function` function with arguments `1` and `2`.
- We then use the `assertEqual` method to assert that the result of the function call is equal to `3`.
- This ensures that the function is working correctly.

6. Looping Structures (Lines 21-25):

```
    def test_list(self):
        my_list = [1, 2, 3]
        for item in my_list:
            self.assertIn(item, my_list)
```

- We define a test method named `test_list`.
- Inside this method, we create a list named `my_list` containing the values `1`, `2`, and `3`.
- We then use a `for` loop to iterate through each item in `my_list`.
- For each item, we use the `assertIn` method to assert that the item is present in `my_list`.
- This ensures that the list contains the expected values.

7. Data Manipulation (Lines 26-30):

```
    def test_string(self):
        my_string = "hello world"
        self.assertEqual(my_string.upper(), "HELLO WORLD")
```

- We define a test method named `test_string`.
- Inside this method, we create a string named `my_string` with the value `"hello world"`.
- We then use the `upper` method on `my_string` to convert it to uppercase.
- We then use the `assertEqual` method to assert that the result of the conversion is `"HELLO WORLD"`.
- This ensures that the string manipulation is working correctly.

8. Control Flow - Part 2 (Lines 31-35):

```
        if my_string.startswith("hello"):
            self.assertTrue(True)
        else:
            self.assertFalse(False)
```

- We use an `if-else` statement to check if `my_string` starts with `"hello"`.
- If it does, we use the `assertTrue` method to assert that `True` is `True`.
- If it doesn't, we use the `assertFalse` method to assert that `False` is `False`.
- This ensures that the string manipulation is working correctly.

9. Function Calls - Part 2 (Lines 36-40):

```
    def test_exception(self):
        try:
```

```
            raise ValueError("This is an error")
        except ValueError:
            self.assertTrue(True)
```

- We define a test method named `test_exception`.
- Inside this method, we use a `try-except` block to handle exceptions.
- We raise a `ValueError` exception with the message `"This is an error"`.
- We then use the `assertTrue` method to assert that `True` is `True` if the exception is caught.
- This ensures that the exception handling is working correctly.

10. Output Generation (Lines 41-45):

```
    def test_output(self):
        print("This is a test output")
```

- We define a test method named `test_output`.
- Inside this method, we use the `print` function to print the message `"This is a test output"`.
- This is used to demonstrate how to generate output in a test case.

11. Error Handling (Lines 46-50):

```
    def tearDown(self):
        pass
```

- We define a `tearDown` method.
- This method is called after each test method is run.
- We leave it empty in this example, but it can be used to clean up any resources or perform any necessary actions after a test method is executed.

Conclusion:

This code demonstrates how to write test cases in Django using the `TestCase` class. It covers various aspects of testing, including user input handling, control flow, function calls, looping structures, data manipulation, exception handling, output generation, and error handling. By understanding this code, beginners and intermediate programmers can learn how to write effective test cases for their Django applications.

File: ml_model.py
Documented code for ml_model.py:
Code Explanation:

1. Import Statements (Line 1):

```
import cv2
from keras.models import load_model
import numpy as np
```

- We import the necessary libraries:
  - `cv2`: OpenCV library for image processing and computer vision.
  - `keras.models`: Keras library for loading pre-trained deep learning models.
  - `numpy`: NumPy library for numerical operations and array handling.

2. Variable Declarations (Line 2-5):

```
# Load the pre-trained model
model = load_model('C:\\Users\\kyath\\OneDrive\\Desktop\\plantrecognition\\plant_recognition_model.h5')
print(model)
```

```

- We load the pre-trained plant recognition model using `load_model`. The model is stored in an HDF5 file.
- We print the model summary to display its architecture and layer details.

**3. Function Definition (Line 6-45):**

```
def plant_recognition_model(image_instance):
    # Read the image from the file-like object
    img = cv2.imdecode(np.frombuffer(image_instance.read(), np.uint8), cv2.IMREAD_COLOR)

    # Resize and preprocess the image
    resized_img = cv2.resize(img, (331, 331)).reshape(-1, 331, 331, 3) / 255.0

    # Make a prediction using the pre-trained model
    prediction = model.predict(resized_img)

    # Define the class names for the prediction
     class_names = ['Karanj Trunk', 'Karanj Leaf', 'Karanj Seed', 'Neem Trunk', 'Neem Leaf', 'Neem Seed', 'Peeple Trunk', 'Peeple Leaf', 'Peeple Seed']

    # Get the predicted class label
    class_prediction = class_names[np.argmax(prediction)]

    # Generate the output text based on the predicted class
    if class_prediction in ['Karanj Trunk', 'Karanj Leaf', 'Karanj Seed']:
        output_text = 'Karanj: Popularly known as Indian Beech in outside India is a medicinal herb used mainly for skin disorders. Karanja oil is applied to the skin to manage boils, rashes, and eczema as well as heal wounds due to its antimicrobial properties. The oil can also be useful in arthritis due to its anti-inflammatory activities.'
    elif class_prediction in ['Neem Trunk', 'Neem Leaf', 'Neem Seed']:
```
```

```
        output_text = 'Neem: A versatile medicinal tree. Neem oil and neem leaves are used for various medicinal purposes. It has anti-inflammatory, antifungal, and antibacterial properties, making it beneficial for skin care, hair care, and managing various health conditions.'
    elif class_prediction in ['Peeple Trunk', 'Peeple Leaf', 'Peeple Seed']:
        output_text = 'Peepal: The bark of the Peepal tree, rich in vitamin K, is an effective complexion corrector and preserver. It also helps in various ailments such as strengthening blood capillaries, minimizing inflammation, healing skin bruises faster, increasing skin resilience, treating pigmentation issues, wrinkles, dark circles, lightening surgery marks, scars, and stretch marks.'
    else:
        output_text = 'Unknown Plant'

    # Return the predicted class label and the output text
    return class_prediction, output_text
```

- We define a function called `plant_recognition_model` that takes an image instance as input.
- Inside the function, we read the image from the file-like object using OpenCV's `imdecode`.
- We resize and preprocess the image to match the input format expected by the pre-trained model.
- We use the loaded model to make a prediction on the preprocessed image.
- We define a list of class names corresponding to the possible plant categories.
- We get the predicted class label by finding the index of the maximum value in the prediction array.
- Based on the predicted class label, we generate an informative output text describing the plant and its medicinal properties.
- Finally, we return the predicted class label and the output text.

4. Function Call (Line 46-50):

```
# Example usage of the function
image_file = open('path/to/image.jpg', 'rb')
class_prediction, output_text = plant_recognition_model(image_file)

# Print the predicted class label and the output text
```

```
print("Predicted Class:", class_prediction)
print("Output Text:", output_text)
```

- We provide an example usage of the `plant_recognition_model` function.
- We open an image file in binary read mode and pass it to the function.
- The function returns the predicted class label and the output text.
- We print the predicted class label and the output text to the console.

Conclusion:

The provided code defines a function, `plant_recognition_model`, that takes an image instance as input and uses a pre-trained deep learning model to predict the plant category. It then generates an informative output text describing the plant and its medicinal properties based on the predicted class label. The function can be used to identify and learn about different plant species from images.