UDACITY

PROJECT SPECIFICATION

Meme Generator

Project Setup and Code Style

CRITERIA	MEETS SPECIFICATIONS
The project demonstrates PEP compliance and programming best practices.	The code adheres to the PEP 8 style guide and follows common best practices, including: • Variable and function names are clear. • The code is DRY (Don't Repeat Yourself) and methods demonstrate the principle of composition.
The project demonstrates an ability to create basic documentation.	All included docstrings and comments adhere to PEP 8 standards A README file is included in each module directory, and each README includes: • The class name. • A brief describing the role-and-responsibilities of the module. • A list of the module's dependencies. • A few examples of how to use the module. • Instructions for setting up and running the program are included in the project root README file.

CRITERIA	MEETS SPECIFICATIONS
The project demonstrates the ability to consume public libraries using virtual environments.	The code makes use of public libraries using a virtual environment. All required dependencies are listed in the root `requirements.txt` file which was created using the `$ pip freeze > requirements.txt` bash command. The program runs with no errors. (Optional) If git is used, the virtual environment directory is added to the .gitignore file.
The project demonstrates implementation of basic Python exception handling.	If the program encounters a common error case (e.g. attempting to load an incompatible filetype), it throws an exception. All exceptions include a human-readable message. (Optional) Make your exception handling even more awesome: • Define custom exception classes for different types of exceptions—for things like *Invalid File, Invalid Text Input (e.g. too long) • Use os.walk to automatically discover ingestible files in a directory
The project demonstrates the ability to create Python modules.	Classes are organized into multiple directories, with related classes being placed together. Each module directory includes a proper `__init__.py` file.

Quote Engine Module

CRITERIA	MEETS SPECIFICATIONS

CRITERIA	MEETS SPECIFICATIONS
The project demonstrates implementation of basic object-oriented data structures.	The code includes a Python class that defines a `QuoteMode` object, which contains text fields for body and author. The class overrides the correct methods to instantiate the class and print the model contents as: `”body text” - author` All related classes are defined in a directory that includes valid `__init__.py` files to declare the package.
The project demonstrates an understanding of when to use Abstract Base Classes (ABC) in Python and the ability to implement the design pattern.	The project contains an abstract base class, `IngestorInterface`, which defines: A complete `classmethod` method to verify if the file type is compatible with the ingestor class. An abstract method for parsing the file content (i.e., splitting each row) and outputting it to a `Quote` object. Only non-abstract classes should be complete. > **Hint:** Classmethods can access class variables, which can be redefined by children classes.
The project demonstrates an ability to ingest text files using the native file library.	The project contains a `TextIngestor` class. The class inherits the `IngestorInterface`. The class does not depend on any 3rd party library to complete the defined, abstract method signatures to parse Text files. The parse method returns a valid `QuoteModel`

CRITERIA	MEETS SPECIFICATIONS
The project demonstrates an ability to ingest DOCX files using the `python-docx` library.	The project contains a `DocxIngestor` class. The class inherits from the `IngestorInterface` class. The class depends on the `python-docx` library to complete the defined, abstract method signatures to parse DOCX files. The parse method returns a valid `QuoteModel`
The project demonstrates an ability to ingest PDF files using CLI tools.	The project contains a `PDFIngestor` class. The `PDFIngestor` class inherits from the `IngestorInterface` class. The `PDFIngestor` class utilizes the `subprocess` module to call the pdftotext CLI utility—creating a pipeline that converts PDFs to text and then ingests the text. The class handles deleting temporary files. The parse method returns a valid `QuoteModel` > NOTE: Do not use the pdftotext PIP Library - we'd like you to demonstrate your understanding of the subprocess module.

CRITERIA	MEETS SPECIFICATIONS
The project demonstrates an ability to ingest CSV files using the `pandas` library.	The project contains a `CSVIngestor` class. The class inherits the `IngestorInterface`. The class depends on the `pandas` library to complete the defined, abstract method signatures to parse CSV files. The parse method returns a valid `QuoteModel`
The project demonstrates an ability to: • implement class inheritance in Python using the *strategy object design pattern* • apply DRY (don't repeat yourself) principles	The `DocxIngestor`, `PDFIngestor`, `CSVIngestor`, and `TextIngestor` classes should realize the `IngestorInterface` abstract base class. Methods that have shared responsibilities are fully defined in the parent class. Excess code is not repeated across the classes. All ingestors are packaged into a main `Ingestor` class. This class encapsulates all the ingestors to provide one interface to load any supported file type.

Meme Generator Module

CRITERIA	MEETS SPECIFICATIONS

CRITERIA	MEETS SPECIFICATIONS
The project demonstrates the ability to use the `Pillow` library to perform basic image operations.	The project defines a `MemeGenerator` module with the following responsibilities: • Loading of a file from disk • Transform image by resizing to a maximum width of 500px while maintaining the input aspect ratio • Add a caption to an image (string input) with a body and author to a random location on the image. The class depends on the `Pillow` library to complete the defined, incomplete method signatures so that they work with JPEG/PNG files. The method signature to make the meme should be: `make_meme(self, img_path, text, author, width=500) -> str #generated image path` The **init** method should take a required argument for where to save the generated images: `__init__(self, output_dir...)`.

Package your Application

CRITERIA	MEETS SPECIFICATIONS

CRITERIA	MEETS SPECIFICATIONS
The project demonstrates an understanding of python arg variables for CLI execution.	The project contains a `main.py` file that uses the `ImageCaptioner`, `DocxIngestor`, `PDFIngestor`, and `CSVIngestor` methods to generate a random captioned image. The program must be executable from the command line. The program takes three OPTIONAL arguments: • A string quote body • A string quote author • An image path The program returns a path to a generated image. If any argument is not defined, a random selection is used.
The project demonstrates an ability to interface with web resources using `flask` and `requests`.	The project completes the Flask app starter code in `app.py`. All @TODO tasks listed in the file have been completed. `app.py` uses the `Quote Engine` module and `Meme Generator` modules to generate a random captioned image. `app.py` uses the `requests` package to fetch an image from a user submitted URL. The flask server runs with no errors

Suggestions to Make Your Project Stand Out!

1. **Make it Unique.** Add your own content (images and quotes).
2. **Unit test everything.** Define unit tests to ensure your code functions as intended.
3. **Deploy as a Webapp.** Deploy the flask server to Heroku so that it can be accessed publicly.
4. **Extend your system.** Be creative by using your meme generator in unique ways – ideas include:

- Sharing the generated image with an email
- Using a 3rd party API to dynamically add more information. You can check out a bunch of 3rd party APIs here. Some possibilities are:
  - Weather, traffic, locations
  - Use Amazon Rekognition to identify the image content and define rules to choose the quote category