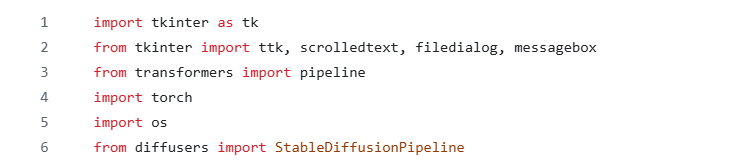
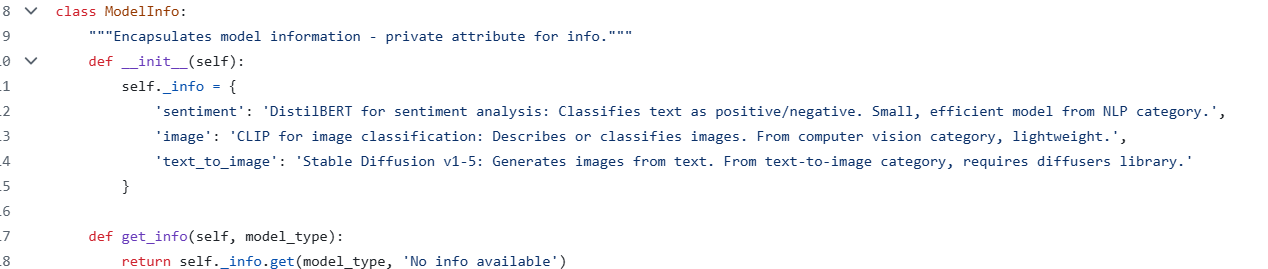
**Code Explanation:**

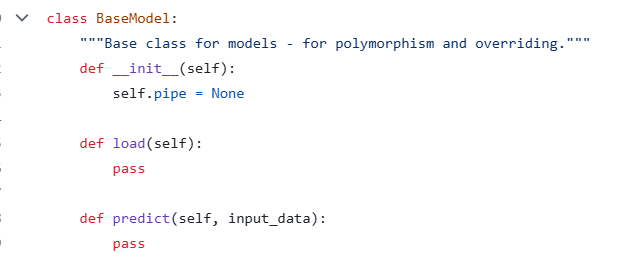
This Python desktop app puts several AI models behind a clean, click-and-go window built with tkinter. Under the hood it uses the transformers and diffusers libraries to do three things: analyze sentiment in text, classify images, and turn text prompts into pictures. The code is arranged to highlight core OOP ideas—encapsulation, inheritance, polymorphism, a logging mixin for multiple inheritance, and a small decorator for button clicks. That structure keeps the project readable and easy to extend, and it also serves as a teaching example of how solid software-engineering practices can bring order to a fairly complex set of features.



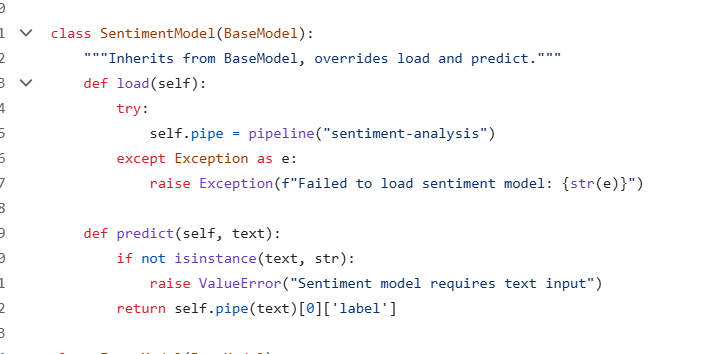
At first we started his program by importing all required libraries. The tkinter family (tk, ttk, scrolledtext, filedialog, and messagebox) forms the GUI layer: it provides widgets, a scrollable text area, file pickers, and user-friendly dialogs for errors and messages. The AI functionality is delivered by Hugging Face’s transformers—exposed through the high-level pipeline interface—and by diffusers for text-to-image generation via StableDiffusionPipeline. Although torch (PyTorch) is not called explicitly, it underpins model computation. We used os for file-system checks such as verifying image paths. Together these imports make it possible to build an interactive desktop app that runs modern pretrained models.



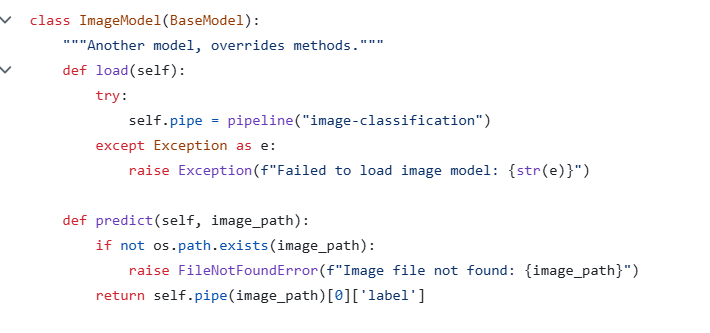
We used different classes in this program. THe first class ModelInfo shows up first in the code. ModelInfo holds the short notes for each model—sentiment, image, and text-to-image—in a private dictionary. Other parts of the program don’t touch that dictionary directly; they call get\_info() to read what they need. That’s classic encapsulation: the data stays hidden while the class exposes a small, stable way to access it.



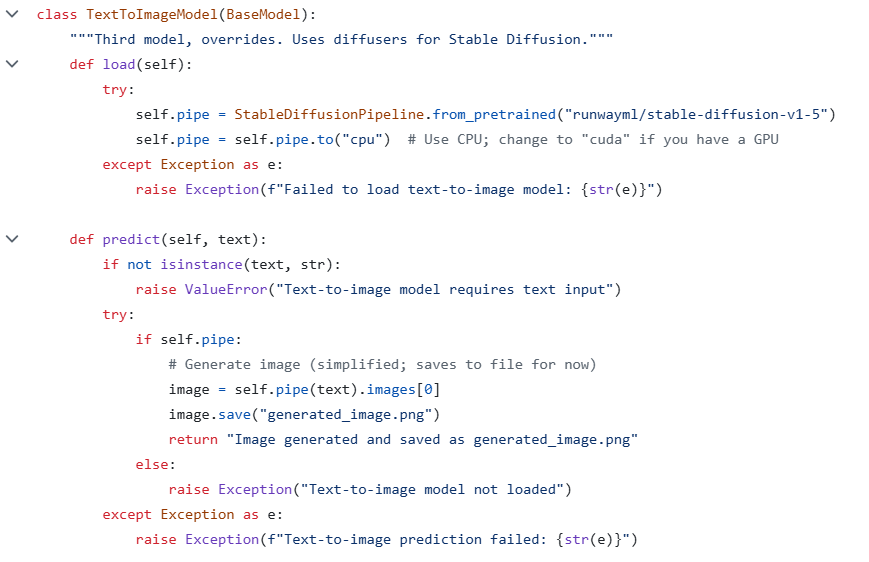
Next comes BaseModel, a lightweight parent that defines the common shape of every model in the app. It provides shared state (pipe) and two methods—load() and predict()—which concrete models override. This gives the GUI one consistent interface, no matter which specific model is in use.. We used the attribute self.pipe later to hold the loaded inference pipeline, while the load and predict methods are intentionally left empty. Subclasses will override these methods to implement task-specific loading and inference. This pattern enables **inheritance** and prepares the ground for **polymorphism**, since the GUI can interact with any model through the same method names. The subclasses—SentimentModel, ImageModel, and TextToImageModel—each provide specific behavior.



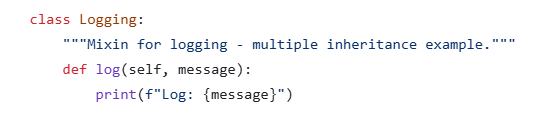
The SentimentModel class wraps Hugging Face’ pipeline("sentiment-analysis"), which usually pulls in a lightweight DistilBERT model fine-tuned for sentiment tasks. Its predict method first checks that the input is actually text, then runs the pipeline and returns the top label (like “POSITIVE” or “NEGATIVE”). If the model can’t be loaded, it raises a clear, user-friendly error instead of crashing—solid defensive programming for a GUI app.



We created the class ImageModel to provide image classification via pipeline("image-classification"). This class typically defaults to a Vision Transformer model. Before it runs any predictions, the predict method makes sure the given path actually points to a file. That simple check saves you from confusing errors later. It then returns the top class label. The flow mirrors SentimentModel, so the interface feels the same no matter whether you’re working with text or images.



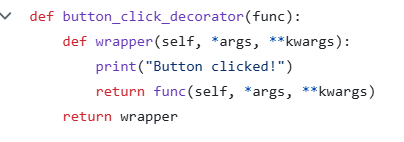
Another class TextToImageModel hooks up Stable Diffusion v1-5 through the diffusers library. Its load step builds the pipeline and, by default, sends it to the CPU so it works on most machines. If a GPU is available, switching the device to "cuda" speeds up image generation significantly.



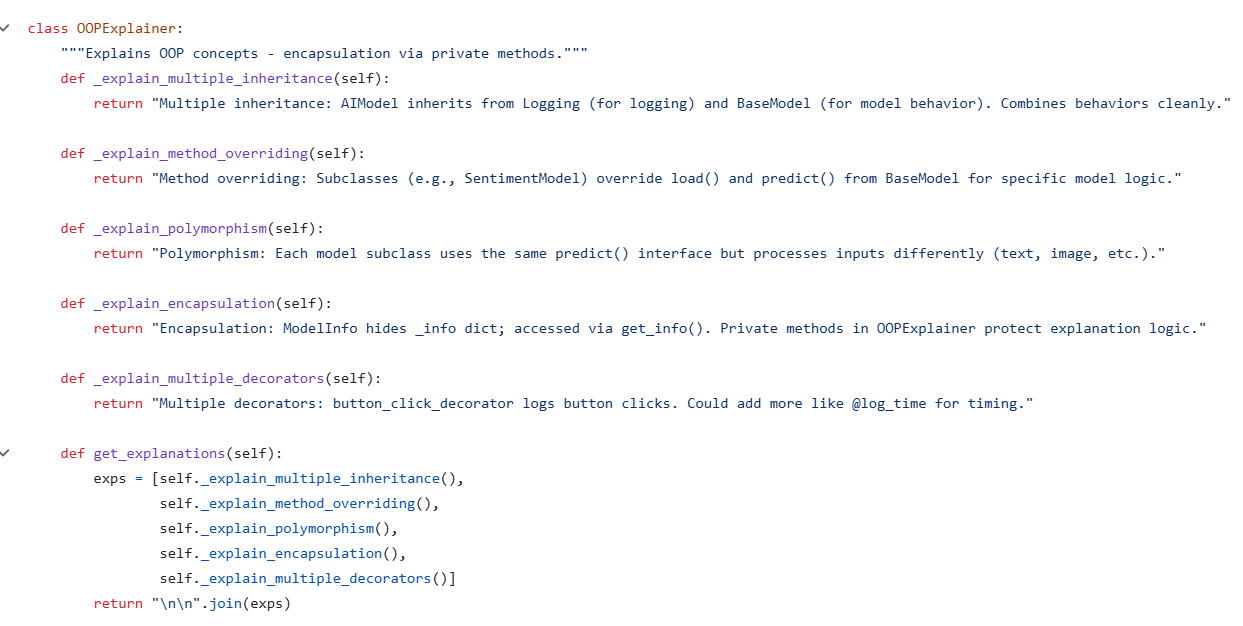
Logging is a small mixin that adds a log() helper. You attach it to other classes with multiple inheritance, so you get console messages without stuffing logging code into the model classes. It’s a simple, Pythonic way to bolt on a shared feature.



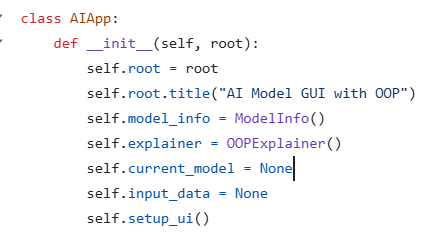
AIModel inherits from both Logging and BaseModel. It acts as a thin wrapper that also chooses which concrete model to use. The constructor takes a string, passes it to \_create\_model, and gets back the appropriate subclass. From there, load() and predict() simply delegate to that inner instance, while the mixin’s log() method prints lightweight status updates—such as confirming that a model has finished loading. By keeping the choice of model in one place, the interface doesn’t need to know about any specific classes. It can just call predict() on AIModel and get a result—whether that’s from text analysis, image classification, or image generation—which is a straightforward example of polymorphism.



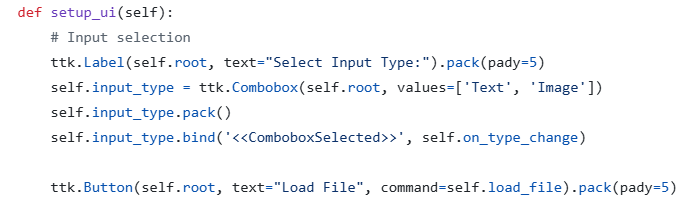
We use a small decorator, button\_click\_decorator, to add a log line to the Run Model action. The decorator runs first, prints “Button clicked!”, and then executes \_decorated\_run\_model(). The handler’s code stays exactly as it is.



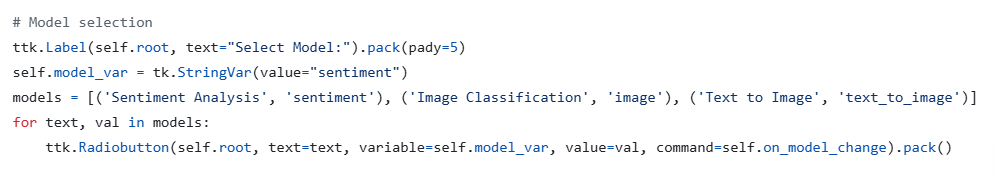
OOPExplainer provides self-contained textual descriptions of the OOP principles embodied in the code. Each explanation method is marked private by convention (single underscore) to indicate they are internal helpers. The get\_explanations method aggregates these texts into a single formatted string. Beyond their didactic value, these explanations become live content displayed in the GUI, linking theory to the concrete implementation the user is operating.



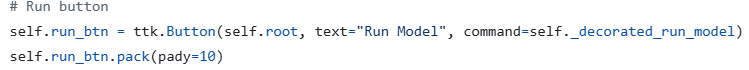
AIApp is the class that runs the window and puts the interface together. Its initializer takes the tkinter root, sets the title, creates helpers like ModelInfo (for model notes) and OOPExplainer (for the teaching blurbs), sets up the state variables current\_model and input\_data, and then calls setup\_ui() to build the widgets. By keeping the startup steps separate from the layout work, the code stays easier to read and maintain.



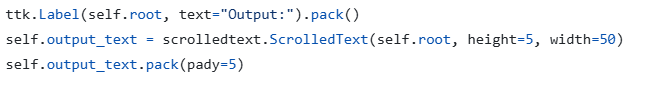
At the start of setup\_ui, the app builds the basics: a label, a drop-down to choose “Text” or “Image,” and a **Load File** button. The drop-down is wired to on\_type\_change(), which clears any previously loaded input when you switch types. That small reset prevents mix-ups—like treating an image path as text—and keeps the workflow tidy.



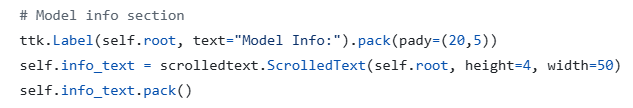
In the next part of setup\_ui, the app adds radio buttons for choosing which model to use. They’re tied to a single StringVar that tracks the current choice. When you click one, on\_model\_change() runs, creates the right AIModel instance, and updates the model info panel. It starts with Sentiment Analysis selected by default so you can run something as soon as the window opens.



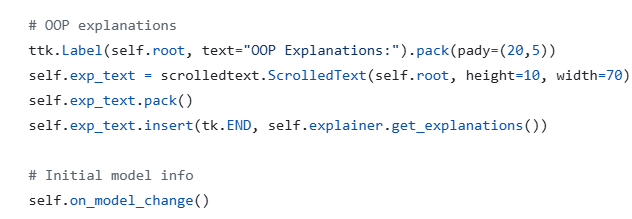
This button invokes the core action: running the selected model on the loaded input. It points to \_decorated\_run\_model, which is wrapped by the earlier decorator to log button clicks. Centralizing execution in one method simplifies validation and error handling.



There is an Output pane—a scrollable text box that shows whatever the model returns. When you run sentiment analysis or image classification, it prints the predicted label; for text-to-image, it confirms that the picture was created and saved. Making this area scrollable keeps long messages manageable now and leaves room for richer output in the future.



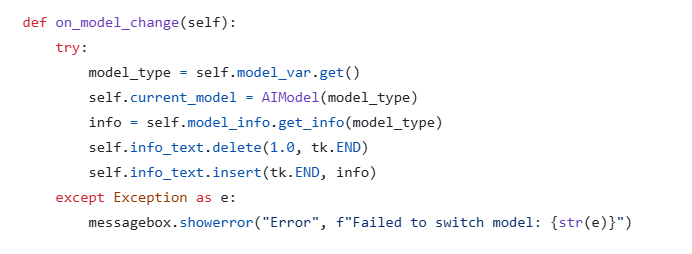
This section presents a human-readable description for the currently selected model. It is populated from ModelInfo via on\_model\_change. Providing inline guidance improves usability and pedagogical clarity, especially for users who are new to these model families.



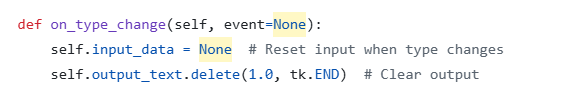
Finally, there is a small panel in the window that displays short notes from OOPExplainer, underscoring that the app is meant to teach as well as run models. During setup, the program calls on\_model\_change() so the default model’s details are already filled in when the window opens, which makes the interface feel informative from the very first click.



This decorated handler orchestrates validation, loading, and prediction. It first verifies that data is loaded and that a model is selected, then enforces modality compatibility (image model requires image input; sentiment and text-to-image require text). Upon passing checks, it loads the model (downloading weights if needed), performs inference, and writes the result to the output area. Any failure is surfaced via a message box rather than a crash, maintaining a smooth user experience.



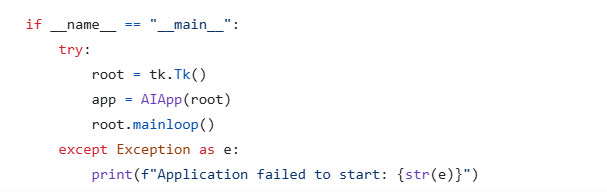
on\_model\_change synchronizes internal state with the user’s radio-button selection. It constructs a fresh AIModel wrapper for the chosen type and refreshes the “Model Info” panel from ModelInfo. Wrapping this flow in a try/except ensures that a failure to initialise a model does not destabilize the interface.



This simple handler resets previously loaded data and clears any displayed result whenever the user switches between text and image input modes. It prevents accidental reuse of an incompatible input and signals clearly that a fresh load is required.



load\_file prompts the user for a file appropriate to the selected input type and then loads it into memory: text files are read into a string; image files are represented by their paths. Errors are handled gracefully, distinguishing missing files from other exceptions and preserving the app’s responsiveness.



The entry point constructs the tkinter root window, instantiates AIApp, and starts the GUI event loop. The startup sequence is wrapped in a try/except so that any initialization failure produces a clear console message rather than an abrupt termination. This final block creates the Tk window, builds the app, and start the event loop. That’s what makes it open as a desktop program when you run the file directly, instead of doing nothing until it’s imported somewhere else.

Stepping back, the project isn’t just a front end for a few models—it’s a compact case study in using OOP on a real task. Encapsulation shows up in the way model details are read through a small, safe interface. Inheritance gives every model the same shape to follow, and polymorphism lets the app call one predict() method even though the work behind it might be text analysis, image classification, or image generation. A small mixin adds logging through multiple inheritance, and a decorator layers in a click message without touching the handler’s body. Put together with a simple GUI and careful error handling, the result is both a practical tool and a clear teaching example of how good structure makes complex features easier to build, use, and extend.