**RAG\_CORPUS.ipynb**

**Cell 1: Install Dependencies:**

****

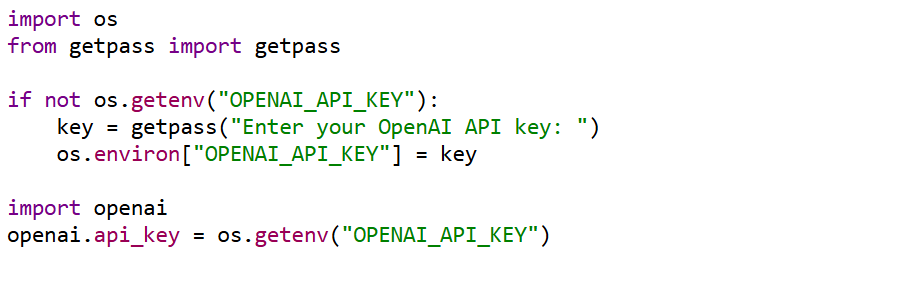
**Purpose:**  
Install all required packages:

* **sentence-transformers** for embeddings
* **faiss-cpu** for nearest‑neighbor search
* **openai** for LLM calls
* **gradio** for the web UI
* **rouge\_score** for evaluation metrics

**Notes:**

* We pin no versions here, assuming latest stable releases.
* The --quiet flag keeps output minimal, making Colab clean.

**Cell 2: Prompt for API Key:**

****

**Purpose:**  
Securely prompt for the OpenAI API key at runtime and set openai.api\_key for subsequent calls.

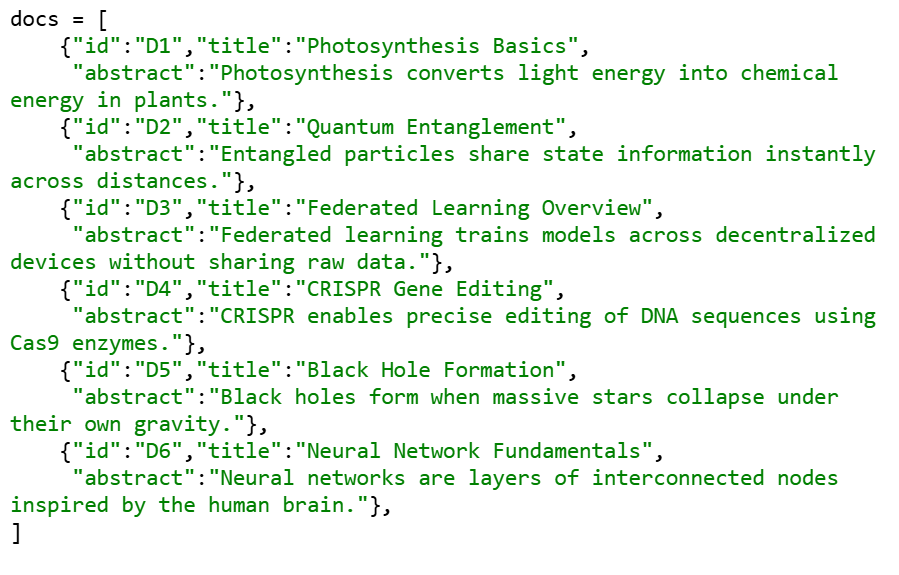
**Key steps:**

1. Check if OPENAI\_API\_KEY is already in environment.
2. If not, ask the user (masked input).
3. Assign it to os.environ and to openai.api\_key.

**Notes:**

* Using getpass avoids echoing the key in notebooks.
* This cell must run before any openai usage.

**Cell 3: Define the Tiny Corpus:**

****

**Purpose:**  
Hard‑codes a 6‑document “toy” corpus covering diverse topics, so we can rapidly prototype without external data downloads.

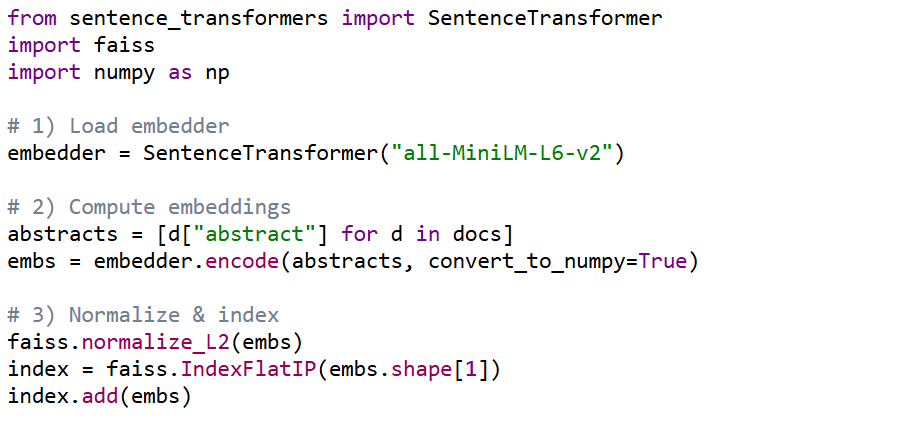
**Inputs / Outputs:**

* **Input:** None (manually defined)
* **Output:** docs list of dicts, each with id, title, abstract.

**Notes:**

* IDs D1–D6 allow consistent reference in retrieval.
* Titles are used later in the UI; abstracts feed into embeddings.

**Cell 4: Embed & Build FAISS Index:**

****

**Purpose:**  
Generate and index semantic embeddings of the 6 abstracts for fast similarity search.

**Key steps:**

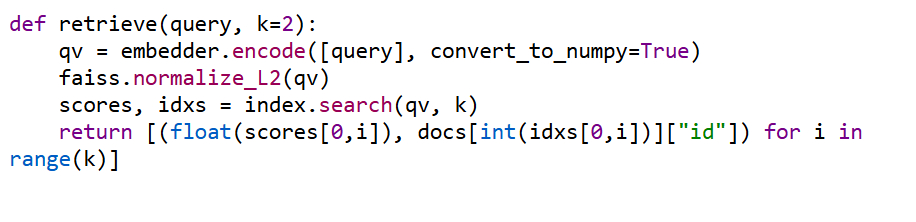
1. Load a lightweight Transformer for sentence embeddings.
2. Encode all abstracts into a NumPy array.
3. Normalize vectors to unit length (so inner product = cosine similarity).
4. Create a FAISS IndexFlatIP (exact search) and add vectors.

**Inputs / Outputs:**

* **Input:** docs list
* **Output:** index ready for .search() queries.

**Notes:**

* IndexFlatIP is chosen for simplicity—no disk persistence needed on tiny data.

**Cell 5: Retrieval Function:  
**

**Purpose:**  
Encapsulates the semantic search step: embed the user query, normalize, and retrieve top‑k matches.

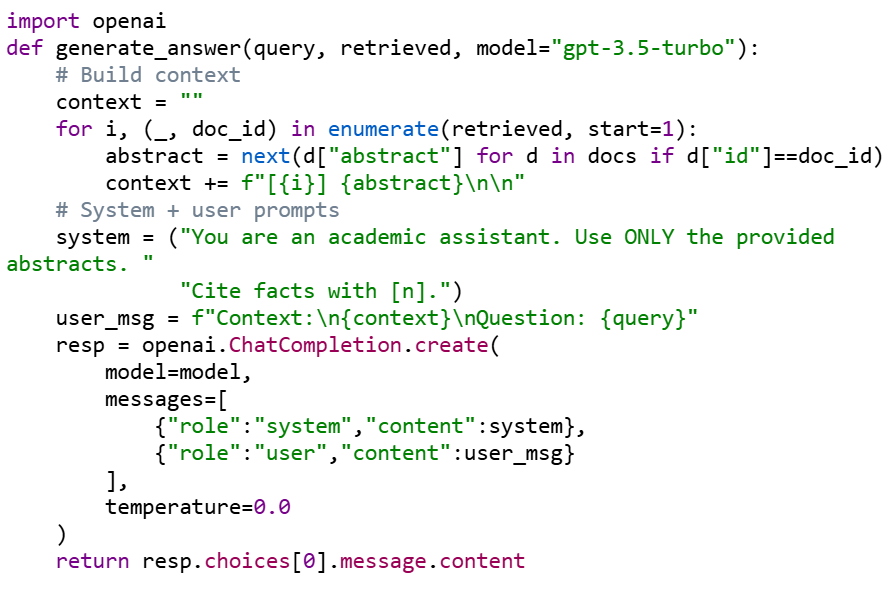
**Inputs / Outputs:**

* **Input:** query (str), k (int)
* **Output:** List of (score, doc\_id) tuples.

**Line‑by‑line:**

* embedder.encode(...) maps query to vector
* Normalization aligns cosine metric
* index.search(...) returns top‑k distances and indices
* We map FAISS index back to docs IDs.

**Cell 6: Prompt Engineering & Answer Generation:**



**Purpose:**  
Constructs a two‑part prompt that feeds the retrieved abstracts as numbered context and instructs the LLM to cite each fact with its source number.

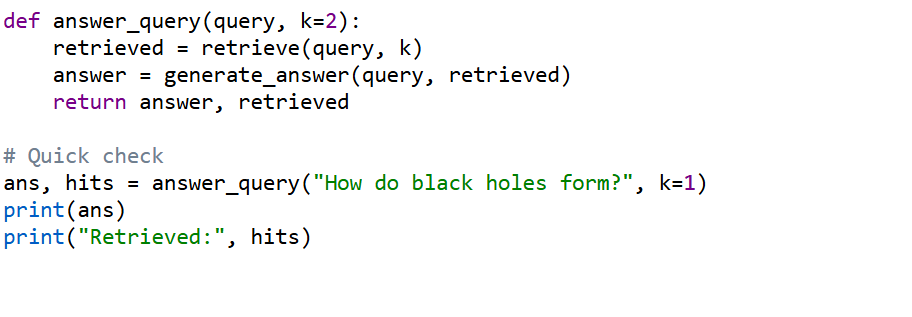
**Key steps:**

1. Iterate retrieved to build a block where each abstract is prefixed by [i].
2. Define a concise system instruction to enforce citation style.
3. Combine with user message containing context + question.
4. Call the OpenAI chat API with temperature=0 for deterministic outputs.
5. Return the LLM’s answer text.

**Notes:**

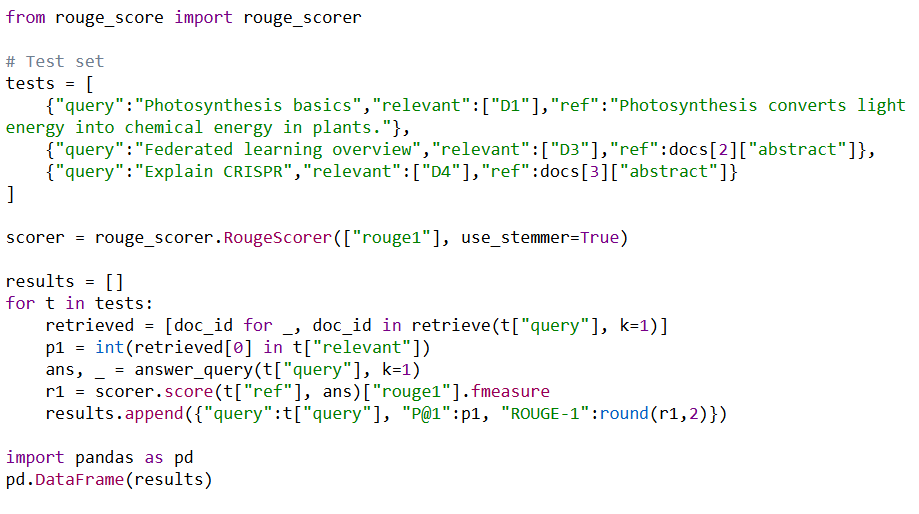
* next(...) lookup over 6 docs is fine; for large corpora you’d map IDs to abstracts in a dict first.

**Cell 7: Wrapper & Quick Test:**

****

* **Purpose:**  
  Combines retrieval and generation into a single callable.  
  The test verifies that the pipeline returns a plausible answer and the expected doc IDs.

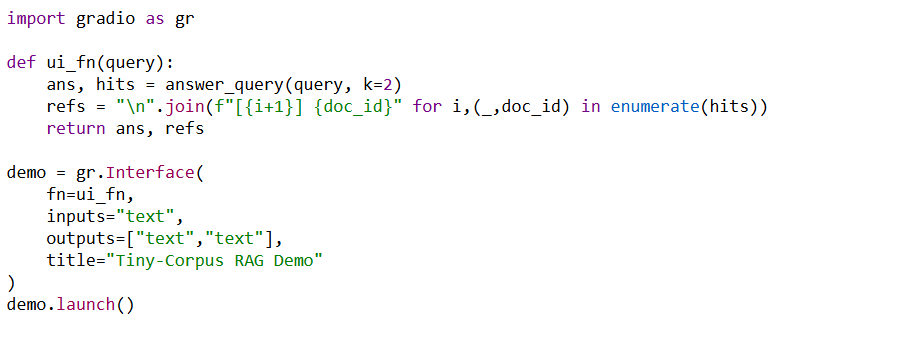
Cell 8: Evaluation Metrics:

****

**Purpose:**

* Defines a small test suite of 3 queries with gold‐standard references.
* Computes **Precision@1** for retrieval.
* Runs the RAG pipeline for each query, computes **ROUGE‑1 F1** between the generated answer and the reference.
* Presents results in a DataFrame.

**Cell 9: Gradio UI (Simple Version):**



**Purpose:**

* Sets up a minimal Interface: a single text box → answer + list of source IDs.
* No chat history, no fancy styling just enough to demo the pipeline.