EXAMPLE 1

Code Explanation

```
python

from transformers import pipeline
```

- **What it does:**
  - Imports the **`pipeline`** function from Hugging Face's **Transformers** library.
  - **`pipeline`** is a **high-level API** that makes it easy to use pre-trained models without worrying about model architecture, tokenization, or preprocessing.
  - It handles everything under the hood: text → tokens → model inference → output.

```
python

# 1) create a sentiment analysis pipeline using a ready-made model
classifier = pipeline("sentiment-analysis",
                      model="distilbert-base-uncased-finetuned-sst-2-english")
```

- **Step 1: Create pipeline**
  - **`pipeline("sentiment-analysis")`** tells Hugging Face you want to build a **sentiment analysis pipeline**.
  - **`model="distilbert-base-uncased-finetuned-sst-2-english"`** specifies the model to use:
    - **`distilbert-base-uncased`**: A smaller, faster, and cheaper version of BERT (Bidirectional Encoder Representations from Transformers).
    - **Fine-tuned on SST-2**: This means the model was trained on the **Stanford Sentiment Treebank v2**, a dataset for positive/negative sentiment classification.
    - **uncased**: Text is lowercased before processing (no difference between "Good" and "good").

👉 Usage: This pipeline will now classify whether a given sentence is **POSITIVE** or **NEGATIVE**, along with a probability score.

```
python

# 2) inference
text = "I loved the new feature — it's super helpful and intuitive!"
```

- **Define input text**
  - This is the sentence we want to analyze for sentiment.
  - It expresses a strong **positive** opinion.

```
python

result = classifier(text)   # returns a list of dicts like [{'label': 'POSITIVE',
'score': 0.999}]
```

- **Run inference**
  - Passes the text to the pipeline.
  - The model tokenizes the text, runs it through **DistilBERT**, and applies a classification head.
  - Output format: a **list of dictionaries** because the pipeline can handle multiple inputs at once.
    Example:

    ```
    python

    [{'label': 'POSITIVE', 'score': 0.999}]
    ```

    - **`label`**: predicted sentiment class.
    - **`score`**: confidence/probability (closer to 1 → higher confidence).

```
python

print(result)
```

- **Display result**
  - Prints the sentiment analysis output for the given text.
  - Expected output (approx.):

    ```
    css

    [{'label': 'POSITIVE', 'score': 0.999...}]
    ```

🔎 Usage of this Hugging Face Model

- **Model:** `distilbert-base-uncased-finetuned-sst-2-english`
- **Task:** Binary **sentiment classification** (POSITIVE / NEGATIVE).
- **Real-world applications:**
  - Analyzing customer feedback (positive vs negative reviews).
  - Social media monitoring (detecting public sentiment about a product, brand, or topic).
  - Chatbots (understanding emotional tone of user input).
  - Business analytics (tracking satisfaction trends).

TEXT GENERATION

Code Explanation

```
python

from transformers import pipeline
```

- **Imports `pipeline`** from Hugging Face's **`transformers`**.
- Just like before, **`pipeline`** is a wrapper that makes it easy to use pre-trained models for common tasks (here: text generation).

```
python

generator = pipeline("text-generation", model="gpt2")
```

- **Create a text generation pipeline**.
  - **`"text-generation"`** → tells the pipeline we want to generate free-form text.
  - **`model="gpt2"`** → loads the **GPT-2 model** (Generative Pretrained Transformer 2).
    - GPT-2 is an **autoregressive language model**: it predicts the next token (word/character piece) given the previous context.
    - Pre-trained on a large corpus of internet text.
    - Unlike DistilBERT (classification), GPT-2 is designed for **generation**.

```
python

prompt = "In a future where healthcare is fully decentralized,"
```

- **Define the seed text** (a "prompt").
- GPT-2 will continue generating text based on this prefix.
- Example: It might talk about futuristic hospitals, AI doctors, or patient privacy — depending on random sampling.

```
python

outputs = generator(
    prompt,
    max_new_tokens=80,
    do_sample=True,
```

```
    temperature=0.8,
    top_p=0.9,
    num_return_sequences=2
)
```

- **Run text generation with settings**:
  - `max_new_tokens=80` → generate up to **80 tokens** (roughly words/subwords).
  - `do_sample=True` → enables **random sampling** (instead of greedy deterministic output).
  - `temperature=0.8` → controls randomness:
    - Lower (<1.0) → more focused, deterministic.
    - Higher (>1.0) → more creative, diverse.
  - `top_p=0.9` (**nucleus sampling**) → instead of picking from all possible next words, only sample from the **smallest set of words whose probabilities sum to 90%**. This reduces nonsensical outputs.
  - `num_return_sequences=2` → generate **2 different completions** for the same prompt.

👉 Together, these settings encourage more natural and varied text.

```
python

for i, out in enumerate(outputs):
    print(f"--- generated {i} ---")
    print(out["generated_text"])
```

- **Loop through generated results**.
- `outputs` is a list of dictionaries, each like:

  ```
  python

  {"generated_text": "In a future where healthcare is fully decentralized, ..."}
  ```

- Prints both generations (with indexes).
- Example output:

  ```
  kotlin


  --- generated 0 ---
  In a future where healthcare is fully decentralized, patients control their
  data...

  --- generated 1 ---
  In a future where healthcare is fully decentralized, blockchain and AI
  collaborate...
  ```

🔎 Usage of This Hugging Face Model

- **Model:** `gpt2` (OpenAI's GPT-2).
- **Task:** Text generation (predicting next words).
- **Capabilities:**
  - Creative writing (stories, dialogue, brainstorming).
  - Autocompletion (finishing partial text).
  - Prototyping chatbots or assistants.
  - Generating synthetic data (caution: may be biased or nonsensical).

⚠️ Limitations:

- GPT-2 is **not fine-tuned for factual correctness**. It can generate plausible but inaccurate or biased information.
- For real-world apps, better options include **GPT-Neo, GPT-J, or GPT-3.5-like open-source models** available on Hugging Face.

QUESTION & ANSWERING

Code Explanation

```
python

from transformers import pipeline
```

- Imports the **pipeline** function from Hugging Face Transformers.
- **`pipeline`** abstracts model loading, tokenization, and inference for a specific task (here: question-answering).

```
python

qa = pipeline("question-answering", model="distilbert-base-uncased-distilled-squad")
```

- **Create a Question Answering pipeline**.
  - **`"question-answering"`** → tells Hugging Face that the task is **extractive QA**: given a context and a question, the model selects a span of text as the answer.
  - **`model="distilbert-base-uncased-distilled-squad"`**:
    - **DistilBERT**: a smaller, faster version of BERT.
    - **Fine-tuned on SQuAD** (Stanford Question Answering Dataset).
    - **uncased** → text is lowercased before tokenization.
  - This model is trained to **extract answers directly from a passage**, not generate free-form text like GPT-2.

```
python

context = (
    "The Apollo 11 mission landed on the Moon on July 20, 1969. "
    "Neil Armstrong and Buzz Aldrin walked on the lunar surface."
)
```

- **Context passage**: the text containing information that the model will search for the answer.
- It can be multiple sentences.

```python
question = "When did Apollo 11 land?"
```

- **Question to ask the model**.
- The pipeline will find the answer **from the context**.

```python
answer = qa(question=question, context=context)
```

- **Run the QA pipeline**.
- Inputs:
  - **`question`**: what we want to know.
  - **`context`**: where the answer may appear.
- Output is a dictionary with:
  - **`'answer'`**: the text span extracted from the context.
  - **`'score'`**: confidence probability of the prediction.
  - **`'start'`** and **`'end'`**: character positions of the answer in the context.

Example output:

```python
{'score': 0.99, 'start': 31, 'end': 44, 'answer': 'July 20, 1969'}
```

```python
print(answer)  # {'score':..., 'start':..., 'end':..., 'answer':'July 20, 1969'}
```

- **Display the model's answer**.
- Here, it correctly extracts **`"July 20, 1969"`** from the context.

🔎 Usage of This Hugging Face Model

- **Model: `distilbert-base-uncased-distilled-squad`**
- **Task: Extractive Question Answering**
- **Applications:**

- Chatbots that answer questions from documents.
- Customer support automation (answering FAQs from manuals).
- Document search and summarization.
- Educational tools (answer questions from textbooks).

⚠️ **Limitations:**

- Only extracts answers **present in the context**.
- Cannot answer questions requiring outside knowledge.
- Accuracy depends on how well the context covers the question.

TEXT SUMMARISATION

Code Explanation

```
python

from transformers import pipeline
```

- Imports the **pipeline** function from Hugging Face Transformers.
- **`pipeline`** provides an easy interface for common NLP tasks (here: summarization).

```
python

summarizer = pipeline("summarization", model="facebook/bart-large-cnn")
```

- **Create a summarization pipeline**.
  - **`"summarization"`** → tells Hugging Face that the task is **abstractive summarization**, where the model generates a concise version of the text.
  - **`model="facebook/bart-large-cnn"`**:
    - **BART** (Bidirectional and Auto-Regressive Transformers) is a sequence-to-sequence model.
    - **Large** → high-capacity model for better performance.
    - **Fine-tuned on CNN/DailyMail** → trained on news articles and summaries.

Usage: This model produces **human-readable summaries** rather than just extracting sentences (contrast with extractive summarization).

```
python

article = (
    "Long article text ... (imagine 400+ words here describing a news story about
climate policy) "
)
```

- **Input text**: the article or document to summarize.
- Can be multiple paragraphs or 100s of words.

```
python

summary = summarizer(article, max_length=80, min_length=25, do_sample=False)
```

- **Generate summary** with specific parameters:
  - **`max_length=80`** → the summary will not exceed 80 tokens (roughly words/subwords).
  - **`min_length=25`** → ensures the summary is at least 25 tokens.
  - **`do_sample=False`** → disables sampling, making the output **deterministic** (uses greedy decoding).

Output:

- **`summary`** is a **list of dictionaries**, each containing a generated summary, e.g.:

  ```
  python

  [{'summary_text': 'Climate policy is evolving...'}]
  ```

```
python

print(summary[0]["summary_text"])
```

- **Print the generated summary text** from the first item in the list.
- Example output:

  ```
  vbnet

  Climate policy is evolving rapidly, with governments implementing new measures to
  reduce emissions and address global warming...
  ```

🔎 Usage of This Hugging Face Model

- **Model: `facebook/bart-large-cnn`**
- **Task: Abstractive summarization**
- **Applications:**
  - Summarizing long news articles or research papers.
  - Creating executive summaries from reports.
  - Assisting content curation and social media posts.
  - Document understanding in enterprise workflows.

MACHINE TRANSLATION

Code Explanation

```
python

from transformers import pipeline
```

- Imports the **pipeline** function from Hugging Face Transformers.
- **`pipeline`** provides a simple interface for NLP tasks, including translation.

```
python

translator = pipeline("translation_en_to_de", model="Helsinki-NLP/opus-mt-en-de")
```

- **Create a translation pipeline**.
  - **`"translation_en_to_de"`** → specifies English-to-German translation.
  - **`model="Helsinki-NLP/opus-mt-en-de"`**:
    - **OPUS-MT** model from Helsinki-NLP.
    - Trained on large parallel corpora of English-German sentence pairs.
    - Uses a **MarianMT** architecture (sequence-to-sequence transformer).

Purpose: Converts English sentences into fluent German translations.

```
python

text_en = "The museum will be closed on Monday for maintenance."
```

- **Input text** in English to be translated.

```
python

output = translator(text_en, max_length=80)
```

- **Run translation**.
- Parameters:
  - **`max_length=80`** → ensures the output does not exceed 80 tokens.

- Returns a **list of dictionaries**, each containing a translated text:

```
python

[{'translation_text': 'Das Museum ist am Montag wegen Wartungsarbeiten
geschlossen.'}]
```

```
python

print(output[0]["translation_text"])
```

- **Print the translated German sentence** from the first dictionary in the list.
- Example output:

```
nginx

Das Museum ist am Montag wegen Wartungsarbeiten geschlossen.
```

🔎 Usage of This Hugging Face Model

- **Model:** `Helsinki-NLP/opus-mt-en-de`
- **Task: Machine translation (English → German)**
- **Applications:**
  - Translating documents, news articles, or website content.
  - Supporting multilingual chatbots or assistants.
  - Assisting in localization workflows.

⚠️ Limitations:

- Works best for general text; domain-specific jargon may be less accurate.
- Output quality depends on the training data (may occasionally produce slightly unnatural phrasing).

SENTENCE EMBEDDING

Code Explanation

```
python

from sentence_transformers import SentenceTransformer, util
```

- Imports:
  - **`SentenceTransformer`** → for loading pre-trained models that encode sentences into dense vectors.
  - **`util`** → provides helper functions, like **cosine similarity**, for comparing embeddings.

```
python

model = SentenceTransformer('sentence-transformers/all-MiniLM-L6-v2')
```

- **Load a pre-trained sentence embedding model**:
  - **`'all-MiniLM-L6-v2'`** is a **compact, efficient transformer** model designed for high-quality sentence embeddings.
  - Converts sentences into **dense numerical vectors** in a semantic space.
- **Usage:** These embeddings can be used for semantic search, clustering, or similarity comparisons.

```
python

sentences = [
    "Hugging Face provides tools for NLP and more.",
    "Hugging Face is a community and platform for ML models."
]
```

- A list of **two sentences** we want to compare for semantic similarity.

```
python

embeddings = model.encode(sentences, convert_to_tensor=True)
```

- **Encode sentences into vectors**:

    - `convert_to_tensor=True` → converts embeddings into PyTorch tensors computation.
- Output: `embeddings` is a tensor of shape `[num_sentences, embedding_dim]`.
    - Each row represents the semantic embedding of a sentence.

```
python

# cosine similarity
cos_sim = util.cos_sim(embeddings[0], embeddings[1])
```

- **Calculate cosine similarity** between two embeddings:
    - Measures **how similar the sentences are in meaning**.
    - Cosine similarity ranges from -1 (opposite meaning) to 1 (identical meaning).

```
python

print("cosine similarity:", float(cos_sim))
```

- **Print the similarity score** as a float.
- Example output (approx.):

    ```
    yaml

    cosine similarity: 0.85
    ```

    - Shows that the two sentences are **highly semantically related**.

🔎 Usage of This Model

- **Model:** `sentence-transformers/all-MiniLM-L6-v2`
- **Task:** Produces **sentence embeddings** for semantic similarity, clustering, or search.
- **Applications:**
    - **Semantic search:** find sentences or documents similar to a query.
    - **Paraphrase detection:** determine if two sentences convey the same meaning.
    - **Recommendation systems:** match items or content based on semantic similarity.
    - **Clustering documents:** group similar text passages automatically.

⚠️ Limitations:

- Embeddings are fixed-length; may lose subtle nuances in very long texts.

LLM INTERACTIONS

Code Explanation

```
python

from huggingface_hub import InferenceClient
```

- Imports **`InferenceClient`** from the Hugging Face Hub Python library.
- This client allows you to **interact with hosted models** on Hugging Face via API, including large language models (LLMs).

```
python

# Choose a model: Mistral or LLaMA
MODEL_ID = "mistralai/Mistral-7B-Instruct-v0.3"
```

- **Specify the model ID** to use for inference.
  - **`"mistralai/Mistral-7B-Instruct-v0.3"`** → an **instruction-tuned 7B-parameter Mistral model**.
  - Designed to **follow instructions in natural language**.
  - Alternative models could include **LLaMA-based models** or other instruction-tuned LLMs.

```
python

# Initialize client
client = InferenceClient(api_key=os.environ["HF_TOKEN"], model=MODEL_ID)
```

- **Create an API client** for inference.
- Parameters:
  - **`api_key=os.environ["HF_TOKEN"]`** → your Hugging Face API token stored in environment variables.
  - **`model=MODEL_ID`** → model to use for inference.
- This sets up authenticated access to the model hosted on Hugging Face.

```python
# Helper to extract plain text
def extract_text(resp):
    try:
        return resp.choices[0].message.content
    except Exception:
        try:
            return resp['choices'][0]['message']['content']
        except:
            return str(resp)
```

- **Utility function** to safely extract the generated text from the API response.
- Handles different response formats:
  - **`resp.choices[0].message.content`** → modern format for chat responses.
  - Fallbacks to older dictionary-style responses or string conversion.

```python
# Example chat
messages = [
    {"role": "system", "content": "You are a helpful programming tutor."},
    {"role": "user", "content": "Explain recursion simply with a Python example."}
]
```

- **Message list for chat-completion**:
  - **`"system"`** → sets the assistant's behavior or personality.
  - **`"user"`** → the actual user prompt or question.
- This is **similar to OpenAI's Chat API structure**.

```python
resp = client.chat_completion(messages=messages, max_tokens=200)
```

- **Call the chat-completion API**:
  - **`messages`** → conversation so far.
  - **`max_tokens=200`** → maximum number of tokens the model will generate.
- Returns a **response object** containing the model's reply.

```python
print(extract_text(resp))
```

- **Print the assistant's reply** in plain text.
- Example output:

```
less

Recursion is when a function calls itself.
Example in Python:
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
print(factorial(5))  # Output: 120
```

🔎 Usage of This Hugging Face Model

- **Model:** `mistralai/Mistral-7B-Instruct-v0.3`
- **Task:** Instruction-following chat, code explanations, Q&A.
- **Applications:**
  - Chatbots or tutoring assistants for programming or general topics.
  - Generating code explanations or examples.
  - Customer support, documentation assistance, or knowledge retrieval.

⚠️ Limitations:

- Requires an API key for hosted inference.
- Token limit applies (max tokens per request).
- Output may occasionally be inaccurate; always verify code or instructions.

Code Explanation

```
python

# Solution Task 1: Friendly tutor, 3+ lines
messages = [
    {"role":"system","content":"You are a friendly programming tutor. Always answer in
at least 3 lines."},
    {"role":"user","content":"Explain Python dictionaries simply."}
]
```

- **Message setup for chat-completion**:
  - **`"system"`** message → instructs the assistant's behavior:
    - Friendly tone.
    - Minimum response length: at least 3 lines.
  - **`"user"`** message → the actual question: "Explain Python dictionaries simply."
- **Purpose:** System instructions guide the model's **style and verbosity**.

```
python

resp = client.chat_completion(messages=messages, max_tokens=200)
```

- **Request the model for a chat completion**:
  - **`messages`** → the conversation so far.
  - **`max_tokens=200`** → restricts the response length to roughly 200 tokens.
- Returns a **response object** with the model's answer.

```
python

print(extract_text(resp))
```

- **Print the extracted text** from the response using the helper function.
- Expected output (example):

```
pgsql

Sure! A Python dictionary is like a real-world dictionary, but it stores data as
key-value pairs.
You can use keys to look up their corresponding values quickly.
```

```
For example: my_dict = {"apple": 3, "banana": 5} lets you access my_d
to get 3.
```

- Notice the assistant follows **the 3-line minimum instruction**.

🔎 Usage Notes

- **Behavior shaping**: The `"system"` role can control:
  - Tone (friendly, formal, humorous).
  - Detail level (concise, multi-line explanations).
  - Specific instructions for formatting or style.
- **Applications**:
  - Personalized tutoring or coaching bots.
  - Guided programming assistance.
  - Instruction-following chat systems with consistent style.

⚠️ **Tip:** If the assistant does not respect multi-line instructions, you can increase `max_tokens` or clarify in the system message.
