

lab_answer

April 3, 2026

1 Lesson 02 Lab — Claude API Setup & First Calls (ANSWER KEY)

This notebook contains the complete working solutions for all three exercises. Each solution includes inline comments explaining the key concepts.

Note for instructors: Share this file only after students have attempted the exercises independently.

1.1 Prerequisites

Before running this notebook:

1. Install dependencies: `pip install anthropic python-dotenv`
2. Create a `.env` file in the same directory as this notebook with:
`ANTHROPIC_API_KEY=sk-ant-api03-your-key-here`
3. Get your key from console.anthropic.com → API Keys

```
[ ]: # Setup cell - run this first
from dotenv import load_dotenv
import os
import anthropic

# load_dotenv() reads the .env file and puts ANTHROPIC_API_KEY into os.environ
# It must be called before anthropic.Anthropic() is constructed
load_dotenv()

# anthropic.Anthropic() automatically reads os.environ['ANTHROPIC_API_KEY']
# No need to pass the key explicitly - keeping it out of your code
client = anthropic.Anthropic()

print("Setup complete. Client type:", type(client))
print("API key loaded:", bool(os.environ.get("ANTHROPIC_API_KEY")))
```

1.2 Exercise 1 — Your First API Call (Solution)

Key points: - `model` and `max_tokens` are both required parameters - `messages` is a list of dicts with `role` ("user" or "assistant") and `content` - The response text lives at `response.content[0].text`
- Token counts are in `response.usage.input_tokens` and `response.usage.output_tokens`

```
[ ]: # Make the API call
# model: which Claude model to use (haiku is fast and cheap - great for dev)
# max_tokens: hard cap on output length (required by Claude, unlike some other
    ↪ APIs)
# messages: list of conversation turns; must start with role="user"
response = client.messages.create(
    model="claude-3-5-haiku-20241022",
    max_tokens=256,
    messages=[
        {"role": "user", "content": "Tell me a fun fact about large language
    ↪ models."}
    ]
)

# response.content is a list (Claude can return multiple content blocks)
# For text responses, we always want index 0, .text attribute
print("Response:")
print(response.content[0].text)
print()

# usage contains token counts - useful for cost estimation and debugging
print("Token usage:")
print(f"  Input tokens: {response.usage.input_tokens}")
print(f"  Output tokens: {response.usage.output_tokens}")
# stop_reason tells us WHY generation stopped:
# 'end_turn' = Claude finished naturally
# 'max_tokens' = hit the max_tokens cap (response was cut off)
print(f"  Stop reason: {response.stop_reason}")
```

1.3 Exercise 2 — Streaming Responses (Solution)

Key points: - `client.messages.stream()` is used as a context manager (with `... as stream:`) - The `with` block ensures the HTTP connection is closed cleanly after iteration - `stream.text_stream` is a generator that yields string chunks (just the text) - `end=""` prevents a newline after each chunk; `flush=True` forces immediate display

```
[ ]: def stream_response(prompt: str) -> None:
    """Stream a response from Claude, printing each chunk as it arrives."""
    # client.messages.stream() opens an SSE (Server-Sent Events) connection
    # The context manager ensures the connection is properly closed when done
    with client.messages.stream(
```

```

    model="claude-3-5-haiku-20241022",
    max_tokens=512,
    messages=[{"role": "user", "content": prompt}]
) as stream:
    # stream.text_stream yields the raw text of each chunk as it arrives
    # end="" prevents print() from appending \n after every chunk
    # flush=True forces Python to write to stdout immediately (no buffering)
    for text in stream.text_stream:
        print(text, end="", flush=True)
    # Add a final newline so the terminal prompt appears on a new line
    print()

# Test the function with a longer prompt to observe the streaming effect
print("Streaming response:")
print("-" * 50)
stream_response("Explain quantum entanglement in simple terms.")
print("-" * 50)

```

```

[ ]: # Optional: compare streaming vs non-streaming latency
import time

prompt = "Write a haiku about APIs."

# Non-streaming: blocks until the entire response is ready
start = time.time()
response = client.messages.create(
    model="claude-3-5-haiku-20241022",
    max_tokens=64,
    messages=[{"role": "user", "content": prompt}]
)
non_stream_elapsed = time.time() - start
print(f"Non-streaming: response appeared after {non_stream_elapsed:.2f}s")
print(response.content[0].text)
print()

# Streaming: first tokens appear almost immediately
print("Streaming (notice first tokens appear faster):")
start = time.time()
stream_response(prompt)
stream_elapsed = time.time() - start
print(f"Streaming: total wall-clock time {stream_elapsed:.2f}s")
print()
print("Key insight: total time is similar, but user perception is very
↳different!")

```

1.4 Exercise 3 — Multi-turn Conversation (Solution)

Key points: - Claude is **stateless** — it has no memory between API calls by itself - We simulate memory by maintaining a **history** list and sending the full list every call - After each assistant reply, we append it to **history** so it's included in the next call - The history must strictly alternate: user → assistant → user → assistant ...

```
[ ]: def chat_loop() -> None:
    """Run an interactive multi-turn conversation with Claude."""
    # history stores the full conversation so far
    # It must be sent with every API call because Claude is stateless
    history = []
    print("Chat with Claude (type 'quit' to exit)\n")

    while True:
        user_input = input("You: ").strip()
        if not user_input:
            continue # skip empty input
        if user_input.lower() == "quit":
            print("Goodbye!")
            break

        # Step 1: Add the user's message to the history
        # This becomes part of the context Claude will see
        history.append({"role": "user", "content": user_input})

        # Step 2: Call the API with the FULL history
        # Sending all previous turns is what gives Claude "memory"
        response = client.messages.create(
            model="claude-3-5-haiku-20241022",
            max_tokens=512,
            messages=history # send everything we've said so far
        )

        # Step 3: Extract the reply text
        assistant_reply = response.content[0].text

        # Step 4: Add the assistant's reply to history
        # This ensures future calls include this turn for context
        history.append({"role": "assistant", "content": assistant_reply})

        # Step 5: Show the reply to the user
        print(f"Claude: {assistant_reply}\n")

    # Run the chat loop
    # Try: "My name is Alex" then "What is my name?" to verify memory works
    chat_loop()
```

```
[ ]: # Bonus: chat loop with a system prompt for a specific persona
def chat_loop_with_persona(persona: str = "You are a friendly Python tutor.") ↵
    ↪-> None:
        """Multi-turn chat with a customisable system prompt."""
        history = []
        print(f"Chatting with persona: '{persona}'")
        print("Type 'quit' to exit\n")

        while True:
            user_input = input("You: ").strip()
            if not user_input:
                continue
            if user_input.lower() == "quit":
                print("Goodbye!")
                break

            history.append({"role": "user", "content": user_input})

            # The system parameter sets the AI's persona and stays constant across ↵
            ↪all turns
            # It is separate from the messages list - not counted as a turn
            response = client.messages.create(
                model="claude-3-5-haiku-20241022",
                max_tokens=512,
                system=persona,    # + persona applied to every call
                messages=history
            )

            assistant_reply = response.content[0].text
            history.append({"role": "assistant", "content": assistant_reply})
            print(f"Claude: {assistant_reply}\n")

# Uncomment to test:
# chat_loop_with_persona("You are a friendly Python tutor. Keep answers concise ↵
    ↪and use code examples.")
```