

## **Developing and running a reasoning model-powered web app on Google Colab: A technical teaching case to demystify generative AI**

**Thomas L. Ngo-Ye**, *Alabama State University, [tngoye@alasu.edu](mailto:tngoye@alasu.edu)*

**Jae Choi**, *Pittsburg State University, [jchoi@pittstate.edu](mailto:jchoi@pittstate.edu)*

### **Abstract**

Generative Artificial Intelligence (AI) is transforming the economy, labor market, and society. Given the importance of generative AI, Information Systems (IS) faculties need to teach students the fundamental technical concepts of generative AI so that they can have a deeper understanding. This study proposes a technical teaching case of developing and running a reasoning model-powered web app on Google Colab. This paper contributes to the IS education discipline in three areas. First, it demystifies generative AI by presenting the behind-scenes code that powers AI. Second, it introduces reasoning models and explains their behavior. Third, the case can be used as a comprehensive technical introduction to generative AI because it discusses many essential concepts, principles, techniques, and best practices of generative AI.

**Keywords:** demystify generative AI, reasoning large language model (LLM), Qwen3-0.6B model, Google Colab, technical teaching case, information system education

### **Introduction**

Since the release of ChatGPT on November 30, 2024, generative AI has gradually caught the general public's and organizations' attention. More and more individuals and businesses are beginning to explore generative AI for personal and professional tasks. In the foreseeable future, with the relentless advancement of AI technology, society and businesses will be profoundly shaped by generative AI. While some IS faculties and students have used generative AI chatbots, such as ChatGPT. Their understanding of generative AI remains at the surface of merely using chatbots. They lack a deeper comprehension of the technical details of how the Large Language Model (LLM) works. Studies have found that entry-level understanding of AI is just a stepping stone, arguing that AI literacy consists of multi-layered steps including “know,” “understand,” “apply,” “analyze,” “evaluate,” and “create” (Ng, Leung, Chu, & Qiao, 2021). One main goal of the current study is to equip students with a more complete level of AI literacy that goes beyond elementary understanding and basic applications. These stages of AI literacy can serve as learning goals for students as well.

This paper presents a technical teaching case of developing and running a reasoning model-powered web app on Google Colab. The Python code reported in the case will demystify generative AI. Students will better understand how LLM works, both technically and in detail. This case will feature a reasoning model, one of the latest technical advancements of LLM. We will introduce students to the working behavior of reasoning models. The case includes extensive discussions of fundamental concepts, principles, techniques, and best practices of generative AI. The case serves as a comprehensive technical introduction to generative

AI. IS faculties and students unfamiliar with the technical aspects of generative AI will significantly benefit from the case and use it as a beginning guide and inspiration to start the exploration.

## Literature review

Given the importance of generative AI, there has been much recent interesting academic research. Analysis of 5.4M Reddit posts reveals generative AI discussions focus on practical use, positive sentiment, distinct subcommunities (e.g., generative vs. image AI), and YouTube as a key resource, with minimal ethical discourse (Mentzer, Price, & Singh, 2024). Based on a systematic literature review and bibliometric study, a recent paper comprehensively investigates the development and evolution of Generative AI and proposes future research directions (Leong, Hew, Ooi, Tan, & Koohang, 2025).

In the university setting, misuse of generative AI is a valid concern. A recent study shows that managing generative AI misuse in higher education hinges on balancing tool quality, detection effectiveness, and sanction severity, providing strategies to uphold integrity while harnessing AI's benefits (Nazareth & Choi, 2025). College students' ChatGPT usage and its impact on learning are critical topics. One study finds that students' ChatGPT use enhances satisfaction and perceived academic performance via task dependence and enjoyment. However, reliability concerns weaken these links, with students noting usefulness alongside inaccuracies and over-reliance (Zhang, Jiang, Qiao, & Li, 2024). To answer the question, "Is ChatGPT Enhancing Youth's Learning, Engagement, and Satisfaction?" another paper reveals that ChatGPT enhances students' intrinsic motivation by fostering autonomy, competence, and relatedness, moderated by AI literacy, extending Self-Determination Theory to AI education (Shah, Mathur, & Kr. Vishnoi, 2024).

One of the most commonly acknowledged use cases of generative AI is assisting programmers to write code. Hence, investigating the role of generative AI in college programming classes is a salient issue. One study finds that programming students' confidence and frustration with generative AI tools vary by learning styles and prior experience, emphasizing tailored educational strategies (Hashmi, Li, Parise, & Shankaranarayanan, 2024). Another paper proposes a generative AI-driven adaptive programming language learning system (three-module architecture) to address high course failure rates, validated through a GPT-3.5-turbo prototype for personalized task generation and feedback (Kwak, Jenkins, & Kim, 2023).

Exploring the use cases of generative AI in teaching and academic professional work, a recent study identifies practical applications of generative AI by evaluating four free chatbots, presents five use cases (image generation, data creation, article retrieval, note transcription, coding assistance), advocates multi-chatbot strategies, addresses privacy concerns, and highlights Google Colab's utility for teaching generative AI (Ngo-Ye & Choi, 2024). The extant IS education literature covers many critical perspectives of generative AI in higher education. However, we have not seen studies explicitly focusing on demystifying generative AI with the behind-scenes code. Moreover, we have not seen studies introducing reasoning models yet. Furthermore, we have not found any studies that can serve as a comprehensive technical guide to generative AI. Therefore, we aim to address the gaps identified above. This paper will propose a technical teaching case to demystify generative AI. We will showcase the development and running of a reasoning model-powered web app on Google Colab.

## Teaching Case

Many people have experimented with the conventional Large Language Model (LLM) in products such as ChatGPT. Conventional LLMs respond to the user's prompt immediately without deep thinking. On the

other hand, reasoning or thinking models take their time to think through before giving the final answer. Reasoning models expose their internal thought processes and monologues to users. This enables users to better comprehend the thinking steps that an LLM takes to reach the answer. Reasoning models often outperform traditional, non-reasoning models in math, coding, science, and complex logic reasoning and problem-solving tasks. However, reasoning models consume more computing resources and take longer to generate the final answer than non-reasoning models. Many users may be unfamiliar with the reasoning model. Therefore, in this case, we focus on showcasing a reasoning model.

A majority of people usually use AI via web apps or mobile apps. When people use AI, the AI models perform AI inference behind the scenes. AI inference is also referred to as testing. The user prompt is fed into the trained AI model as input during the inference or testing stage. The trained AI model generates a corresponding response as an output. In this case, we intend to introduce Generative AI to business school students. Since an AI-powered chatbot is the most commonly used AI app, we teach students how to build and run an AI-driven question-and-answer web app. We expose the behind-the-scenes technical details to students to demystify AI. Therefore, students will gain hands-on knowledge of loading an AI model and performing AI inference. Throughout the presentation of this case, we explain the fundamental ideas and terminology of generative AI.

Moreover, we identify many areas in the current project that may be further extended to incorporate advanced AI capabilities and features to match the state-of-the-art AI chatbots. Discussing these potential extensions will introduce students to many essential concepts of generative AI, principles, technologies, and best practices.

### **Why choose Qwen3-0.6B as the LLM for the case?**

We select a recently released open-source Large Language Model (LLM), Qwen3-0.6B. It was released on the Huggingface platform (<https://huggingface.co/>) at the end of April 2025. Due to the rapid advancement speed of generative AI since ChatGPT was released on 11/30/2022, newly released AI models tend to improve performance over older models. Therefore, we prefer to experiment with the latest AI models when everything else is equal.

Qwen is a well-established open-source AI model family. Thanks to its usable capability and full spectrum of model size offerings, Qwen is well-received by both the open-source community and organizations that want to deploy open-source AI models on their premises. There are lots of derivative models and adaptors fine-tuned based on Qwen models. Going forward, Qwen will gain more traction with its expanding ecosystem. The letter "B" in the Qwen3-0.6B model means a billion parameters. 0.6B describes the number of parameters of the LLM. From <https://huggingface.co/Qwen/Qwen3-0.6B>, we find that the Qwen3-0.6B model has 752M or 752 million parameters.

Typically, LLMs with a larger number of parameters outperform LLMs with a smaller number of parameters in output quality and accuracy. Large LLMs tend to be smarter than smaller LLMs because large LLMs can hold more knowledge with more parameters than smaller LLMs. Much knowledge from the vast training data is transformed and compressed into LLMs during pre-training. The connections of the deep artificial neural network of LLMs (the weights and biases of the parameters) encode the knowledge. Larger LLMs tend to have stronger capabilities, such as instruction following, language understanding, reasoning, generalization, and emerging capabilities, than smaller LLMs. However, running larger LLMs typically requires higher GPU hardware requirements, a larger amount of VRAM, and a longer time to run.

According to <https://huggingface.co/chat/settings/Qwen/Qwen3-235B-A22B>, the Qwen3-235B-A22B model is the flagship model in the Qwen3 series model family. The Qwen3-235B-A22B model's

performance rivals that of leading reasoning models such as OpenAI o1 and Deepseek R1. The Qwen3-0.6B is the most miniature version in the Qwen3 family.

The reasons for choosing the Qwen3-0.6B model are as follows:

1. It is from a recently released open-source model family with a well-established track record and an expanding ecosystem.
2. It is a capable LLM, given its small size.
3. It is a reasoning model that displays its thinking process in its output. It is beneficial for students to see the thinking process of AI.
4. The small size enables a relatively fast response speed for class demonstration and student interaction with AI.
5. We face the practical hardware limit of the Google Colab free tier, which only offers Nvidia Tesla T4 GPU with 15 GB Video Random Access Memory (VRAM) for free. Larger models demand more powerful GPUs and more VRAM, which are not free.

### **Why choose the Google Colab free tier for the case?**

An AI model typically requires advanced computing resources, primarily GPU and lots of VRAM, to run. Businesses can lease GPU-powered servers from cloud computing vendors or provision their GPU-powered servers and deploy them on-premise. In a higher education environment, we lack the financial budget to lease online GPUs or purchase GPU-powered workstations to deploy on-premise. Moreover, most computers in colleges and universities do not have discrete GPUs essential for running AI models.

In essence, Google Colab is a cloud-based Jupyter Notebook service provided by Google for free. Users can write and execute Python programs in a web-based interactive environment on Google Colab. Fortunately, Google kindly offers the general public limited free online GPUs via its Colab service. In this case, we use the free tier Google Colab to run an LLM. Google Colab offers users a free virtual machine with a T4 GPU and 15 GB of VRAM. Each session on Colab corresponds to a separate logical virtual machine. When a session ends, all the data on that virtual machine is perished. However, users can save their Python-based notebooks to their Google Drive. Users can also download their notebooks as Python programs to a local computer.

Python is one of the most popular computer programming languages for AI. We typically need to download and install Python software on a local computer to run a Python program. Setting up a Python environment for an AI project and installing various required Python packages (dependencies) are cumbersome. Luckily, Google Colab provides the Python 3 virtual environment. Thus, end users do not need to be concerned about installing Python software and setting up a Python environment for an AI project. We can access the Python 3 environment directly through Google Colab Notebook.

### **Why incorporate Gradio web GUI into the case?**

While we can run an LLM in Google Colab notebook and see the AI's response in the notebook output section, the notebook user interface and interaction are not user-friendly. To enhance the user experience, we incorporate the Gradio web interface component in our Python program. Thus, we wrap the LLM inference process in a user-friendly web interface, where users can enter a text prompt and see an AI response.

### **Procedure for setting up and running a reasoning LLM-powered web app on Google Colab**

Next, we present the procedure for setting up and running a reasoning LLM on Google Colab with a Python notebook.

1. First, in a web browser, type "<https://colab.research.google.com/>" and hit the enter key on the keyboard.
2. Second, click the "Sign in" button on the right-top corner of the web page. Enter the user's Gmail as the username and Gmail password as the password.
3. Third, click the "File" menu and select "New notebook in Drive" to create a new Python notebook in your Google Drive.
4. Fourth, click the "runtime" menu and select the "Change runtime type."
5. Fifth, under "Hardware accelerator," select "T4 GPU," and click the "Save" button.
6. Sixth, copy the following Python program source code and paste it into the new notebook. Users can paste with the "Ctrl + V" key combo on a Microsoft Windows PC or the "command + V" key combo on an Apple Mac computer.

The content of Python program is shown below.

```
# This script sets up and runs the Qwen3-0.6B language model with Gradio
# When using CUDA, configure PyTorch's memory allocation behavior to reduce fragmentation:
%env PYTORCH_CUDA_ALLOC_CONF=garbage_collection_threshold:0.9,max_split_size_mb:512
# Install the latest version of Gradio to build a web interface
!pip install --upgrade gradio

# Import necessary modules from Hugging Face Transformers and Gradio
from transformers import AutoModelForCausalLM, AutoTokenizer
import gradio as gr

def predict(prompt):
    """
    Generate text from a user prompt using the Qwen3-0.6B causal language model.
    Steps:
    1. Load model and tokenizer from the pretrained checkpoint.
    2. Format the prompt into the chat template for the model.
    3. Tokenize inputs and run text generation.
    4. Separate out the model's "thinking" (internal logits debug) from the final content.
    5. Return a combined, Markdown-formatted string.
    """
    # Specify the Hugging Face model repository
    model_name = "Qwen/Qwen3-0.6B"

    # Load the tokenizer to convert text <-> token IDs
    tokenizer = AutoTokenizer.from_pretrained(model_name)

    # Load the causal language model with automatic device and dtype mapping
    model = AutoModelForCausalLM.from_pretrained(
        model_name,
        torch_dtype="auto", # Let Transformers choose float16 or float32
        device_map="auto"   # Distribute layers across available devices (CPU/GPU)
    )

    # Prepare a single chat message as expected by the model's chat template
    messages = [
        {"role": "user", "content": prompt}
    ]

    # Apply the chat template to assemble the prompt text, including internal thinking tokens
    text = tokenizer.apply_chat_template(
        messages,
        tokenize=False,      # We will tokenize in the next step
```

```
        add_generation_prompt=True, # Include generation marker for model inference
        enable_thinking=True      # Explicitly set to engage in the thinking mode. Keep special tokens for separating "thinking"
    )

    # Tokenize the prompt text and move tensors to the model's device
    model_inputs = tokenizer([text], return_tensors="pt").to(model.device)

    # Generate new tokens up to a large limit (max_new_tokens)
    generated_ids = model.generate(
        **model_inputs,
        max_new_tokens=32768 # Cap on number of tokens model can output
    )

    # Extract only the newly generated token IDs (skip input prompt length)
    output_ids = generated_ids[0][len(model_inputs.input_ids[0]):].tolist()

    # Attempt to locate the special token index that splits "thinking" vs final content
    try:
        # 151668 is the ID of the thinking delimiter token in this model's vocabulary
        index = len(output_ids) - output_ids[::-1].index(151668)
    except ValueError:
        # If delimiter not found, assume no separate thinking portion
        index = 0

    # Decode tokens before delimiter as the model's internal "thinking"
    thinking_content = tokenizer.decode(
        output_ids[:index], skip_special_tokens=True
    ).strip("\n")

    # Decode tokens after (and including) delimiter as the final answer
    content = tokenizer.decode(
        output_ids[index:], skip_special_tokens=True
    ).strip("\n")

    # Format both parts using Markdown headers for clarity in the UI
    final_output = (
        f"***Thinking:**\n{thinking_content}\n\n"
        f"***Content:**\n{content}"
    )

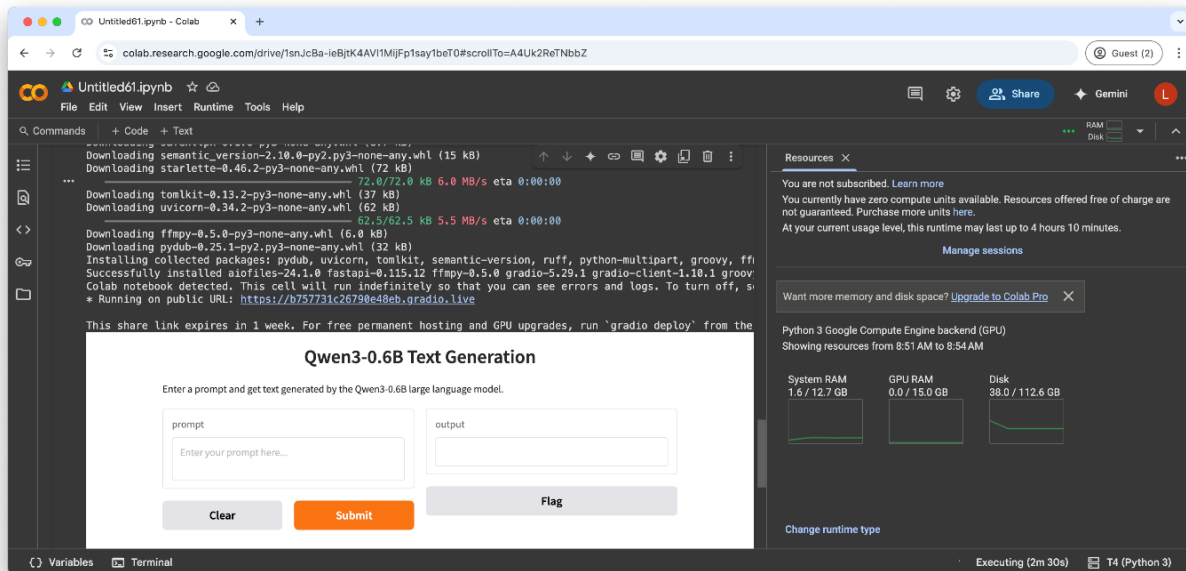
    return final_output # Return combined string to Gradio interface

# Create a Gradio interface to expose `predict` as a web app
iface = gr.Interface(
    fn=predict, # Function to call on user input
    inputs=gr.Textbox(
        lines=2,
        placeholder="Enter your prompt here..."
    ),
    outputs="text", # Display the result as text
    title="Qwen3-0.6B Text Generation",
    description=(
        "Enter a prompt and get text generated by the Qwen3-0.6B large language model."
    )
)

# Launch the Gradio app when running this script directly
if __name__ == "__main__":
    iface.launch(
        share=True, # Create a public URL (helpful for Replit/Colab)
        debug=True, # Enable verbose error tracebacks during development
        server_name="0.0.0.0", # Bind to all network interfaces for external access
        server_port=7860 # Default Gradio port (matches Colab's and ngrok's mapping)
    )
```

7. Seventh, click the "runtime" menu and select "Run all" to execute the entire Python program.

It takes about 35 seconds to complete the initial run of the Python program. The process involves installing the Gradio package and downloading and loading the reasoning LLM, Qwen3-0.6B, into the system memory of the virtual machine on the Google Colab. Users should be patient while waiting for the initial run. A larger model, such as a 7B LLM (B stands for billions of parameters), may take many minutes to complete the initial run.



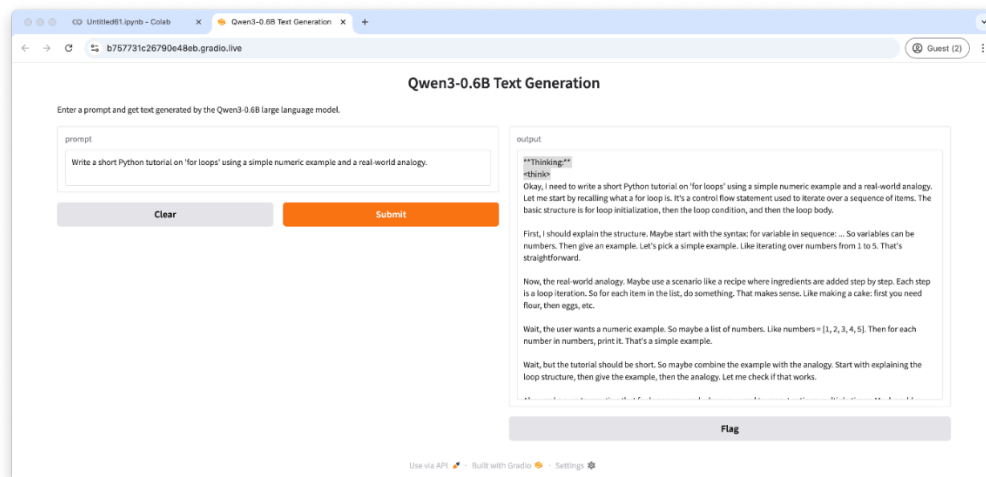
**Figure 1: Result of Running Python Program on Google Colab before LLM Inferencing.**

As illustrated in Figure 1, a web GUI is a user interface generated by the Python program. Inside the Google Colab notebook output, users can enter a text prompt and click the "Submit" button to see the text output. However, for a better user experience, users can click the Gradio-generated temporary public URL in the notebook output section. A new web browser tab will be opened with the public URL, where users can interact with the AI model. Gradio produces a temporary public URL for this AI-power web app in this run. Figure 1 shows the information: "Running on public URL: <https://b757731c26790e48eb.gradio.live>". The bottom right hand corner of Figure 1 shows the usage of Google Colab virtual machine resources (System RAM 1.6/12.7 GB, GPU RAM 0.0/15.0 GB, and Disk 38.0/112.6 GB) after the initial run of the Python program,

System RAM refers to the conventional memory used by the CPU. GPU RAM refers to the VRAM, a critical aspect affecting the performance of AI inference. When the Python program completes its initial run (about 35 seconds), GPU RAM usage is 0 / 15.0 GB because GPU is unnecessary during that running stage. At this moment, we have not entered any text prompt into the AI-power web app. Therefore, the LLM inference has not yet happened. GPU and VRAM will be needed for LLM inference later.

Next, we click the temporary public URL: <https://b757731c26790e48eb.gradio.live> from the Google Colab Notebook output. A new web browser tab opens that allows us to enter the following text prompt in the new tab of Gradio web GUI (shown in Figure 2): "Write a short Python tutorial on 'for loops' using a simple numeric example and a real-world analogy."

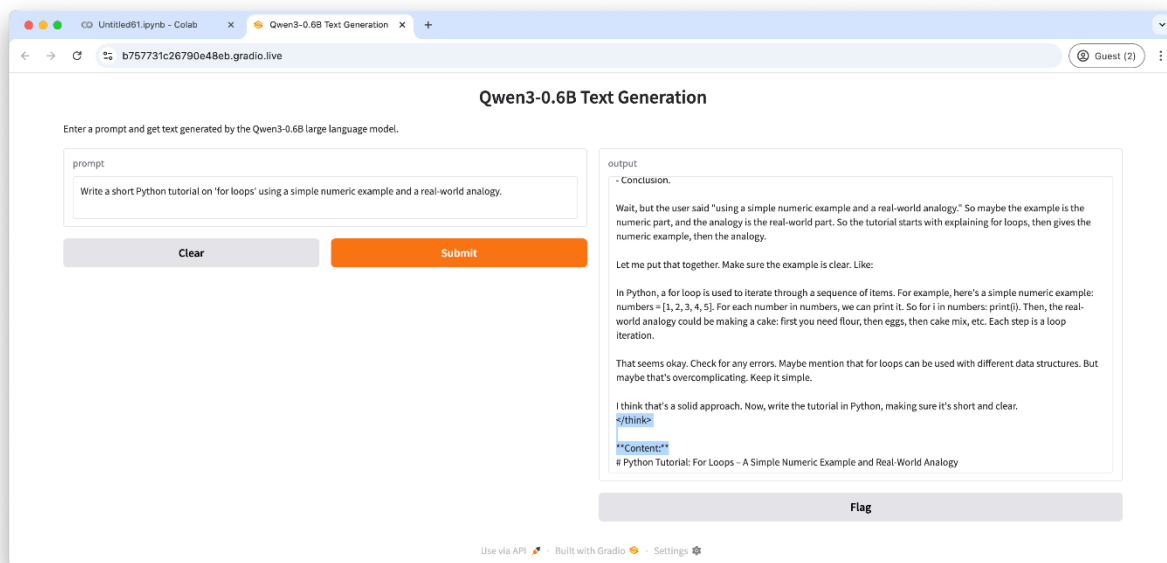
Then, we click the Submit button. We waited about 88.8 seconds before the AI's response was presented in the web GUI.



**Figure 2: Web GUI of AI Text Generation App Using Gradio Part 1.**

Figure 2 shows the "Thinking" segment of the reasoning model output presented at the beginning. The special tag `<think>` marks the start of the thinking portion. Next, we scroll down the AI response to reach the end of the thinking segment.. Figure 3 presents another segment of the reasoning model's response. The tag `</think>` marks the end of the "Thinking" segment. After the tag `</think>`, it presents the regular final output for end users.

The AI output is long, including the thinking segment and the regular output. Figures 2 and 3 only present parts of the whole AI response. Next, we report the entire AI response.



**Figure 3: Web GUI of AI Text Generation App Using Gradio Part 2.**



The entire AI response:

**\*\*Thinking:\*\***

<think>

Okay, I need to write a short Python tutorial on 'for loops' using a simple numeric example and a real-world analogy. Let me start by recalling what a for loop is. It's a control flow statement used to iterate over a sequence of items. The basic structure is for loop initialization, then the loop condition, and then the loop body.

First, I should explain the structure. Maybe start with the syntax: for variable in sequence: ... So variables can be numbers. Then give an example. Let's pick a simple example. Like iterating over numbers from 1 to 5. That's straightforward.

Now, the real-world analogy. Maybe use a scenario like a recipe where ingredients are added step by step. Each step is a loop iteration. So for each item in the list, do something. That makes sense. Like making a cake: first you need flour, then eggs, etc.

Wait, the user wants a numeric example. So maybe a list of numbers. Like numbers = [1, 2, 3, 4, 5]. Then for each number in numbers, print it. That's a simple example.

Wait, but the tutorial should be short. So maybe combine the example with the analogy. Start with explaining the loop structure, then give the example, then the analogy. Let me check if that works.

Also, make sure to mention that for loops are used when you need to repeat actions multiple times. Maybe add a note about when to use them. But since it's a tutorial, maybe keep it concise.

I should avoid too much technical jargon. Use simple language. Maybe start with a heading, then the example, then the analogy, then a conclusion. Let me structure it:

- Introduction to for loops.
- Syntax with example.
- Real-world analogy.
- Conclusion.

Wait, but the user said "using a simple numeric example and a real-world analogy." So maybe the example is the numeric part, and the analogy is the real-world part. So the tutorial starts with explaining for loops, then gives the numeric example, then the analogy.

Let me put that together. Make sure the example is clear. Like:

In Python, a for loop is used to iterate through a sequence of items. For example, here's a simple numeric example: numbers = [1, 2, 3, 4, 5]. For each number in numbers, we can print it. So for i in numbers: print(i). Then, the real-world analogy could be making a cake: first you need flour, then eggs, then cake mix, etc. Each step is a loop iteration.

That seems okay. Check for any errors. Maybe mention that for loops can be used with different data structures. But maybe that's overcomplicating. Keep it simple.

I think that's a solid approach. Now, write the tutorial in Python, making sure it's short and clear.

</think>

**\*\*Content:\*\***

# Python Tutorial: For Loops – A Simple Numeric Example and Real-World Analogy

## **\*\*Introduction to For Loops\*\***

A 'for' loop in Python is used to iterate over a sequence of items, such as numbers, strings, or lists. It allows you to repeat a block of code multiple times as the sequence progresses.

### **--### \*\*Simple Numeric Example\*\***

Here's a basic example:

```
```python
numbers = [1, 2, 3, 4, 5]
for number in numbers:
    print(number)
```
```

This will print:

```
```
1
2
3
4
5
```
```

### **### \*\*Real-World Analogy\*\***

Imagine making a **cake** step by step:

1. **Flour**: First, you need flour.
2. **Eggs**: Then eggs.
3. **Cake Mix**: Next, you add cake mix.
4. **Baking**: Finally, bake the cake.

Each step is a loop iteration. You repeat this process for each item in the sequence.

### **### \*\*Conclusion\*\***

For loops are powerful tools for iterating over sequences and performing repetitive actions. They make it easy to build programs that work smoothly with data.

Let me know if you'd like to extend this with more examples!

We examine the output of the Qwen3-0.6B model and find that, for the most part, the output makes sense. The results of the Qwen3-0.6B model are usable. Therefore, the quality of the Qwen3-0.6B model output is acceptable.

## **Technical discussion**

The fundamental mechanism of LLMs is to predict the most likely token given the context that is fed into LLMs. The token is usually randomly selected from a group of tokens with relatively high probability. Therefore, the prediction of the next token is non-deterministic. Thus, even if we submit the same prompt to the exact LLM multiple times, the LLM gives a slightly different answer each time. We usually do not expect the same answer from an LLM when asking the same question many times. This non-deterministic nature of LLMs, in particular, and generative AI generally differs from traditional computer programs, where the output is deterministic when the input is fixed.

### Context window length

In some rare cases, users of AI chatbots notice that AI forgets the information mentioned in the early rounds of conversation in a chat session. This happens mainly because many rounds of long conversations and much data, such as attachments, are fed into the AI model.

Technically, what happens is that the amount of data from the user prompts and AI responses exceeds the AI model's context window length. In that case, the AI model throws away data far back and appears to forget information early in the chat session. The AI model's context window can be considered short-term working memory, which stores information from the user prompts and AI responses in the current chat session. The context window is the span of text (including the user's current prompt, the user's previous prompts in this chat session, and AI's responses in this chat session) that an AI model can see, read, remember, process, and consider at a single time.

The context window length is the maximum amount of tokens the AI model can handle as input in one go. Different AI models have different context window lengths. Longer context windows allow the AI model to maintain longer, more coherent conversations, summarize longer documents, and generate longer text and computer program code.

According to <https://huggingface.co/Qwen/Qwen3-0.6B>, Qwen3-0.6B's Context Length is 32,768 or 32K. According to <https://huggingface.co/Qwen/Qwen3-235B-A22B>, Qwen3-235B-A22B's Context Length is 32,768 natively and 131,072 tokens with YaRN. According to <https://huggingface.co/google/gemma-3-27b-it>, Google's open-source gemma-3 model family has a total input context window of 128K tokens for the 4B, 12B, and 27B sizes models and 32K tokens for the 1B size model. Google gemma-3 model family has a total output context of 8192 tokens. The Google Gemini model family is proprietary and not open-source. The context window length of Gemini 2.5 Pro is 1 million tokens.

The maximum context length that an LLM supports may not be the ideal context length used in reality. Maxing out the context length will consume a lot of GPU VRAM. In many cases, users cannot access the vast amount of VRAM required to take full advantage of the maximum context length. Running with context length more than VRAM can handle may lead to unexpected system crashes. Moreover, as the context length grows, the computation complexity quadruples due to the underlying mechanism of attention calculation. Thus, the inference speed will be slowed down a lot. Therefore, it is recommended to be cautious when setting context length. It is a good idea to start small and experiment with small increments.

### How to switch between Thinking and Non-Thinking Mode

We can switch between Thinking and Non-Thinking Mode by configuring the following setting in the Python program:

**`enable_thinking=True or enable_thinking=False`**

The model will engage in the thinking or reasoning mode if the above setting is explicitly set to true or leave it as the default value in the `tokenizer.apply_chat_template`.

### Gaps between our project and the state-of-the-art AI chatbots

We explore the mainstream AI chatbots, such as ChatGPT, and learn about their new features beyond the basic text generation function. The rudimental text generation function takes the users' text prompt as input and creates text response as output. Compared to the state-of-the-art AI products, our current AI web app's user experience and capability have many gaps. Exploring the gaps offers students the teachable moments of learning how the current state of AI advances beyond the baseline presented in the project. Next, we will explore the gaps and elaborate on the principles and potential techniques that can implement the features to close the gaps.

## AI output streaming vs. non-streaming

We waited more than a minute before the AI web app presented the output, including both AI's internal thought process and the final regular output. During the waiting period, the Gradio web GUI only presents a timer showing the seconds passed. Users mostly see a big blank textbox with nothing to show up. This user experience is subpar because users must wait until the output is completed. Then, the whole response is displayed simultaneously. This above mode is called non-streaming.

On the other hand, most LLM chatbots today stream AI response. In other words, LLM chatbots return responses chunk-by-chunk in real time. Streaming output improves user experience by reducing perceived latency and giving tangible feedback faster.

In the future, we will enhance the program by enabling streaming output. One possible approach is to utilize the package "TextIteratorStreamer." We need to add the following line of code to the Python program.

```
from transformers import TextIteratorStreamer
```

Moreover, we can build a Streamer based on TextIteratorStreamer to yield generated text token-by-token. We can start model generation in a separate thread to stream the output. We iterate over tokens as they become available from the streamer. Then, we yield the current text collected so far, streaming it out to Gradio web GUI.

## Gauging the speed of the reasoning model-powered web app response

Typically, the speed of an LLM is measured by the number of tokens per second. We observed that the Gradio web GUI displayed 88.8 seconds before showing the entire LLM output. So, 88.8 seconds is the amount of time the LLM spent on inferencing. We have stored the generated text, including the thinking segment. To determine the number of tokens for this text, we use the OpenAI web app <https://platform.openai.com/tokenizer>. We paste the Qwen3-0.6B generated text into the above website.

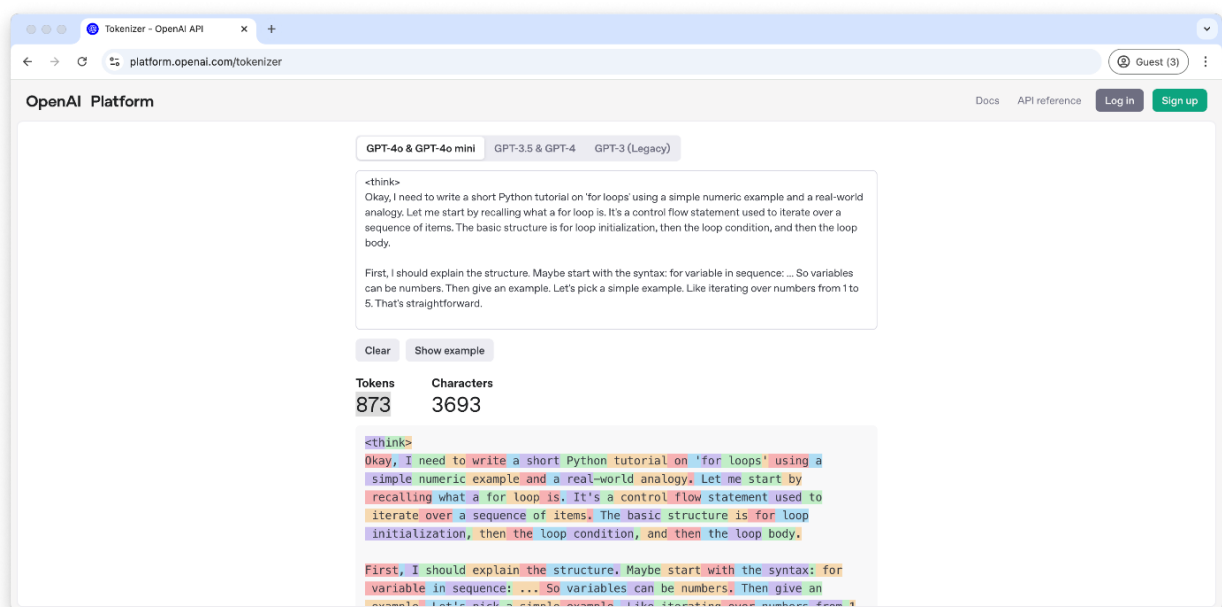


Figure 4: Use OpenAI Tokenizer to Determine the Number of Tokens

Figure 4 shows that the Qwen3-0.6B generated text has 873 tokens. To calculate the speed of Qwen3-0.6B in terms of tokens per second, we use the formula  $\text{speed} = (\text{number of tokens}) / (\text{time})$ .

$$873 / 88.8 = 9.831 \text{ tokens/second}$$

Therefore, when running the Qwen3-0.6B model in the default setting on Google Colab free tier with T4 GPU with 15 GB VRAM, the inference speed is about 9.831 tokens/second. Figure 4 also shows the default setting using GPT-4o & GPT-4o mini's tokenizer. Qwen3-0.6B model's tokenizer differs from GPT-4o and GPT-4o mini. However, for the sake of simplicity, we use GPT-4o & GPT-4o mini's tokenizer to estimate the number of tokens generated by Qwen3-0.6B. Therefore, the token number 873 is just an approximation.

### *Why is inference speed slow for this AI app?*

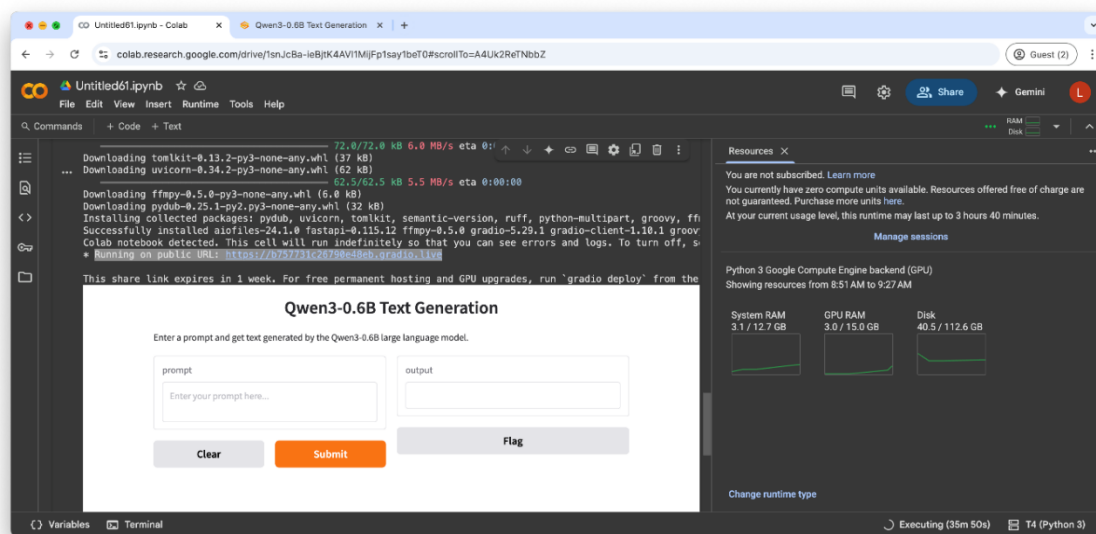
In general, a faster inference speed is preferable. Humans' reading speed and tolerance of latency vary. This 10 tokens/second speed is acceptable for some users if the AI response is delivered in the streaming mode. There are many potential directions to improve the speed of LLM inference, such as using more advanced hardware, optimizing software algorithms, caching common segments of prompts, and batching multiple prompts in a group and submitting them as a batch.

A fundamental block of the working of LLMs is the attention mechanism used in transformer models of LLMs. Flash Attention optimizes the traditional attention mechanism to speed up computations and reduce memory usage while maintaining accuracy. Currently, there are three versions of Flash Attention: V1, V2, and V3. Flash Attention is a viable method to improve AI inference speed. However, the T4 GPU offered in the free tier of Google Colab only might work with Flash Attention V1 with limitations. T4 GPU does not support more advanced Flash Attention V2 and V3.

The hugging Face Accelerator package can be used for both AI model training and AI model inference. Accelerator supports distributed inference, allowing users to efficiently deploy trained AI models across multiple devices (GPUs, CPUs, and TPUs) and run inference tasks over them. Accelerator could speed up AI inference speed. We will explore using Accelerator with the T4 GPU offered in the free tier of Google Colab.

Qwen3-0.6B model is a reasoning model. Generally, a reasoning model's computational complexity is larger than traditional non-reasoning models of the same size. The reason is that a reasoning model needs more intricate processing and attention mechanisms than a non-reasoning model. Due to higher computational complexity, reasoning models require more computer processing power and computation time. Therefore, reasoning models tend to be slower than non-reasoning models. This may explain the relatively low inference speed of 10 tokens/second.

The T4 GPU was released by Nvidia in 2018. Used primarily in data centers, T4 has the relative advantage of energy efficiency compared to the state-of-the-art consumer-grade GPU, such as Nvidia RTX 5090. Nvidia RTX 5090 was released in early 2025. However, due to its outdated architecture, low speed, and bandwidth of its VRAM, the T4 GPU is vastly outperformed by modern GPUs, such as the Nvidia RTX 5090. Therefore, the aged hardware of the T4 GPU contributes to our AI app's relatively low inference speed.



**Figure 5: Result of Running Python Program on Google Colab after LLM Inferencing.**

Figure 5 shows the usage of Google Colab virtual machine resources after we submit a text prompt to the LLM. In other words, it shows resource usage after LLM inferencing (System RAM 3.1/12.7GB GPU RAM 3.0/15.0 GB, Disk 40.5/112.6 GB). The LLM inference process requires GPU work. After the response is completed, we find GPU RAM usage is 3.0 / 15.0 GB. The above resource usage data indicates that for a relatively small-sized reasoning model, such as the Qwen3-0.6B, the System RAM and GPU RAM (VRAM) are not the bottleneck. GPU itself, T4, in this case, is the constraint for AI inference speed.

## Benefits of the project for students

The hands-on text generation project provides several benefits for students. It allows students to gain first-hand experience loading a reasoning LLM on a computer with a GPU and conducting AI inference. We will explain the meaning of the Python code to help students understand the fundamental principles and workflow of LLM inference. This teaching case provides educational value to students and teachers who are unfamiliar with the work of LLM. This case also showcases the behavior and output of a reasoning model. It would be an eye-opening experience for many students.

## Conclusions

This study contributes to the IS education literature in the following ways. First, it demystifies generative AI by presenting the behind-scenes code and the process of running an LLM. Users can see a model's loading process into a computer and the change in GPU usage rate before and after AI inferencing. Second, this study introduces reasoning models and their behaviors. The paper presents the outputs of a reasoning model and tells readers how to spot the delineating markers of the thinking portion of the AI output. Third, we extensively discuss fundamental concepts, principles, techniques, and best practices for generative AI in the teaching case. We compare our hands-on project with the state-of-the-art AI chatbots to identify the gaps. We frame the project's future extensions to close the gaps and motivate the discussion. This paper serves as a comprehensive practical technical introduction to LLMs.

This teaching case targets IS faculty and students unfamiliar with the technical aspects of generative AI. The paper helps them ramp up speed with the current development of LLM. With the program code, the

procedure of running an LLM on Google Colab, and elaborations of the project extensions documented in this study, IS faculty and students can use this teaching case as a stepping stone and inspiration to explore generative AI further.

A series of measurements of AI literacy is the immediate next step, following the framework illustrated in a study on AI literacy (Ng, Leung, Chu, & Qiao, 2021). Achievements in AI literacy can be measured using data derived from learning goals and the dimensions of AI literacy. The comparison of AI literacy across different tools will provide evidence of effectiveness for the technical teaching case reported in this study.

We acknowledge that this study has some limitations. Due to financial constraints, we rely on free cloud-based GPU resources like Google Colab free tier for the hands-on project. Google Colab is assessable for faculty and students. However, the Google Colab free tier has obvious limitations in providing outdated hardware and GPU time constraints. We might explore the opportunity to apply for external grants to teach generative AI in the future. With financial support, we can pay the rental fee for higher-grade cloud-based GPUs to teach AI. Alternatively, we can purchase computers equipped with high-end consumer-grade GPUs like Nvidia GeForce RTX 5090 Graphic Card and install them locally for teaching AI.

## References

- Hashmi, N., Li, Z., Parise, S., & Shankaranarayanan, G. (2024). Generative AI's impact on programming students: frustration and. *Issues in Information Systems*, 25(3), 371-385. doi: [https://doi.org/10.48009/3\\_iis\\_2024\\_128](https://doi.org/10.48009/3_iis_2024_128)
- Kwak, M., Jenkins, J., & Kim, J. (2023). Adaptive programming language learning system based on. *Issues in Information Systems*, 24(3), 222-231. doi: [https://doi.org/10.48009/3\\_iis\\_2023\\_119](https://doi.org/10.48009/3_iis_2023_119)
- Leong, L.-Y., Hew, T.-S., Ooi, K.-B., Tan, G. W.-H., & Koohang, A. (2025, April 1). Generative AI: Current Status and Future Directions. *Journal of Computer Information Systems*, 1–34. doi:<https://doi.org/10.1080/08874417.2025.2482571>
- Mentzer, K., Price, J., & Singh, J. (2024). Analyzing reddit discourse surrounding generative AI. *Issues in Information Systems*, 25(3), 277-292. doi:[https://doi.org/10.48009/3\\_iis\\_2024\\_122](https://doi.org/10.48009/3_iis_2024_122)
- Nazareth, D. L., & Choi, J. (2025, April 22). Deterrence and Generative Artificial Intelligence Use in Universities. *Journal of Computer Information Systems*, 1–15. doi:<https://doi.org/10.1080/08874417.2025.2492205>
- Ng, D., Leung, J., Chu, S., & Qiao, M. (2021). Conceptualizing AI literacy: An exploratory review. *Computers and Education: Artificial Intelligence*, 2(100041). doi:<https://doi.org/10.1016/j.caeai.2021.100041>
- Ngo-Ye, T. L., & Choi, J. (2024). Practical use cases of generative AI in teaching and academic. *Issues in Information Systems*, 25(1), 434-449. doi:[https://doi.org/10.48009/1\\_iis\\_2024\\_135](https://doi.org/10.48009/1_iis_2024_135)
- Shah, C. S., Mathur, S., & Kr. Vishnoi, S. (2024, August 2). Is ChatGPT Enhancing Youth's Learning, Engagement and Satisfaction? *Journal of Computer Information Systems*, 1–16. doi:<https://doi.org/10.1080/08874417.2024.2380698>

Zhang, X., Jiang, H., Qiao, Z., & Li, P. (2024, August 8). Students' Response to ChatGPT: An Adaptive Technology-To-Performance Model. *Journal of Computer Information Systems*, 1–18.  
doi:<https://doi.org/10.1080/08874417.2024.2386546>