

Assistants API Overview (Python SDK)



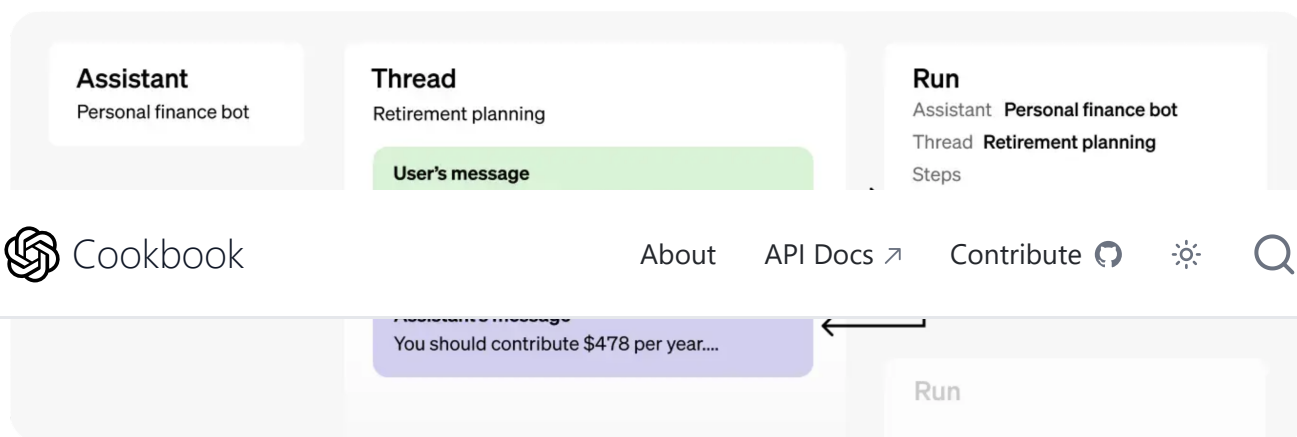
Ilan Bigio

Nov 10, 2023



Open in Github

The new **Assistants API** is a stateful evolution of our **Chat Completions API** meant to simplify the creation of assistant-like experiences, and enable developer access to powerful tools like Code Interpreter and Retrieval.



Chat Completions API vs Assistants API

The primitives of the **Chat Completions API** are Messages , on which you perform a Completion with a Model (gpt-3.5-turbo , gpt-4 , etc). It is lightweight and powerful, but inherently stateless, which means you have to manage conversation state, tool definitions, retrieval documents, and code execution manually.

The primitives of the **Assistants API** are

- Assistants , which encapsulate a base model, instructions, tools, and (context) documents,
- Threads , which represent the state of a conversation, and

- `Runs` , which power the execution of an `Assistant` on a `Thread` , including textual responses and multi-step tool use.

We'll take a look at how these can be used to create powerful, stateful experiences.

Setup

Python SDK

“Note We've updated our Python SDK to add support for the Assistants API, so you'll need to update it to the latest version (1.2.3 at time of writing).”

```
!pip install --upgrade openai
```



And make sure it's up to date by running:

```
!pip show openai | grep Version
```



```
Version: 1.2.3
```



Pretty Printing Helper

```
import json

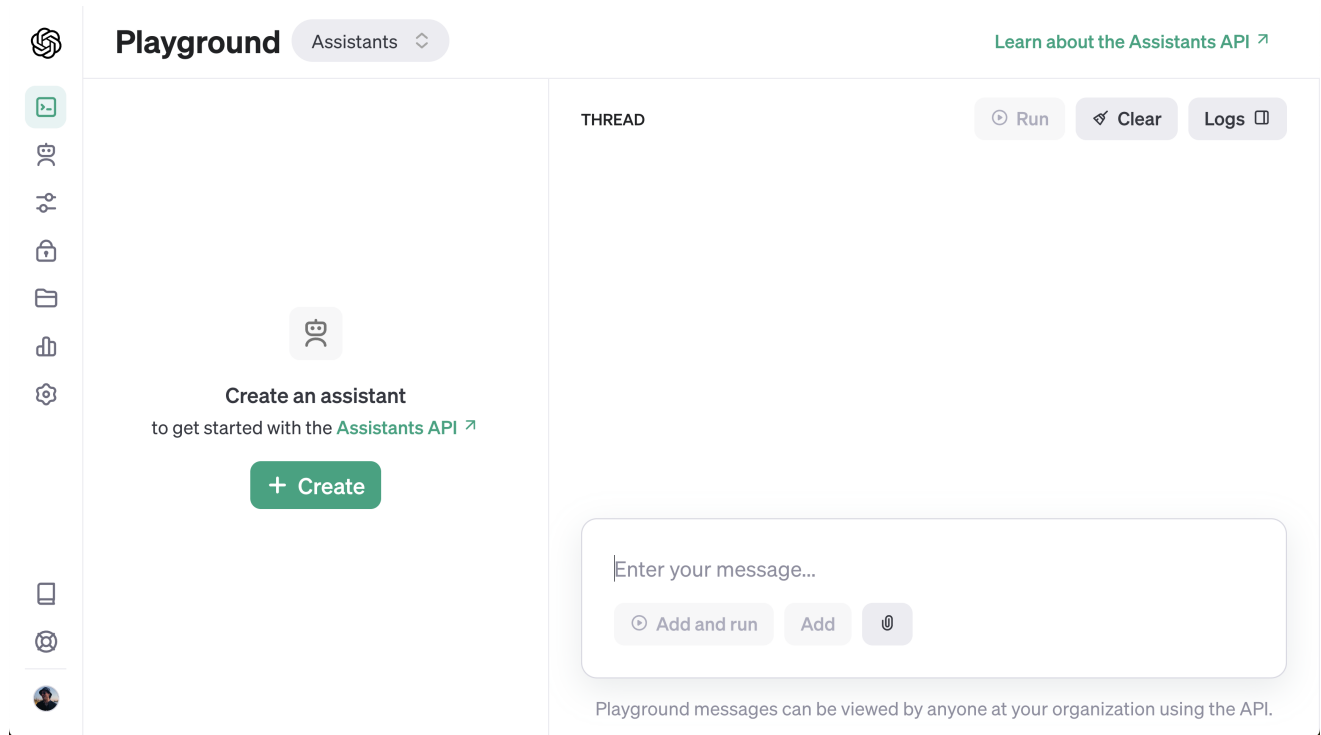
def show_json(obj):
    display(json.loads(obj.model_dump_json()))
```



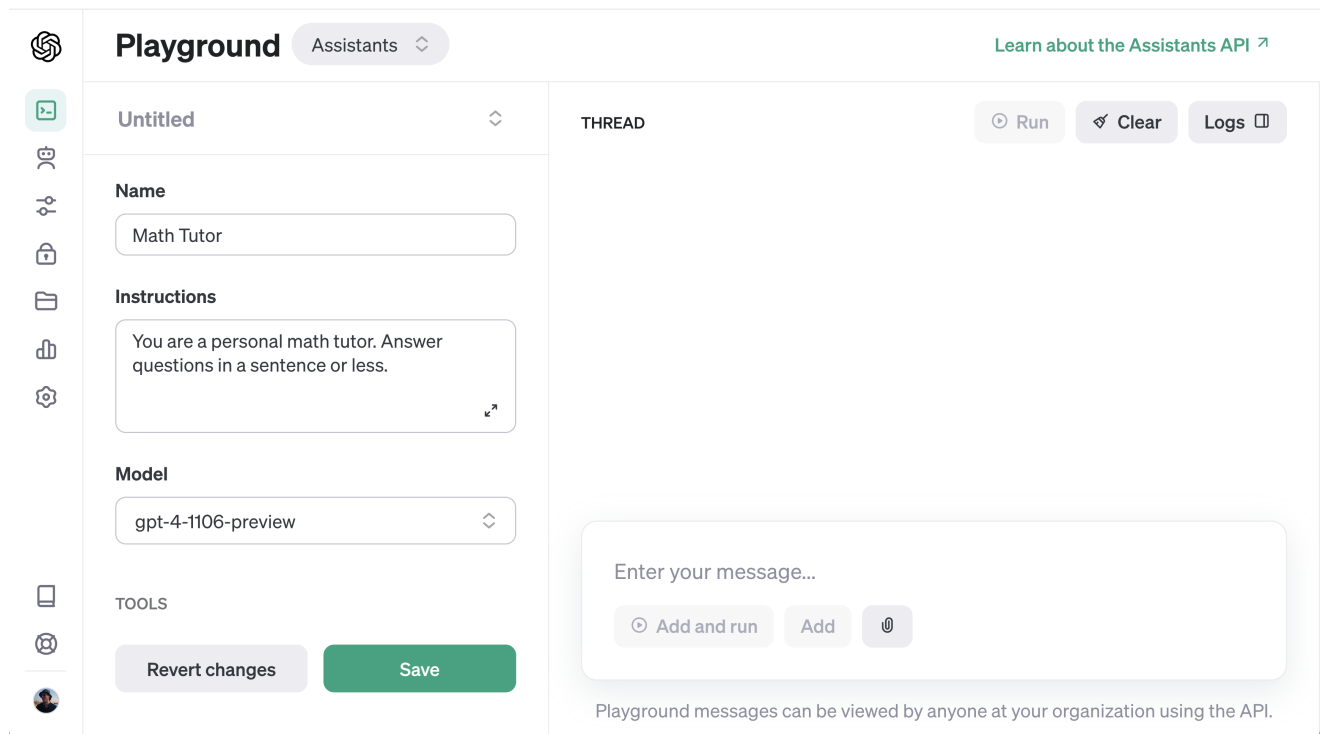
Complete Example with Assistants API

Assistants

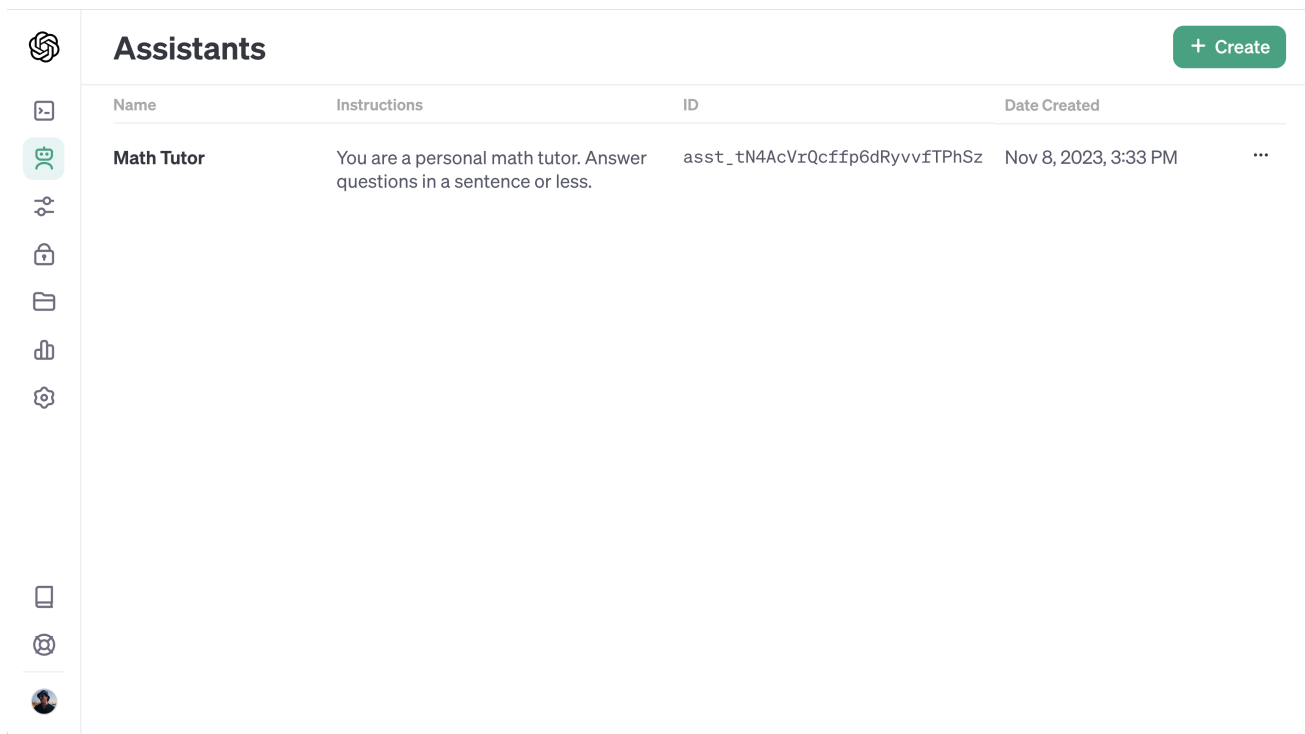
The easiest way to get started with the Assistants API is through the [Assistants Playground](#).



Let's begin by creating an assistant! We'll create a Math Tutor just like in our [docs](#).



You can view Assistants you've created in the [Assistants Dashboard](#).



The screenshot shows the OpenAI Assistants Dashboard. On the left is a sidebar with icons for Assistants, Files, and Settings. The main area is titled "Assistants" and features a "+ Create" button in the top right. Below the title is a table with the following columns: Name, Instructions, ID, and Date Created. A single assistant is listed with the name "Math Tutor", instructions "You are a personal math tutor. Answer questions in a sentence or less.", ID "asst_tN4AcVrQcfff6dRyvvtPhSz", and creation date "Nov 8, 2023, 3:33 PM". A three-dot menu icon is visible to the right of the row.

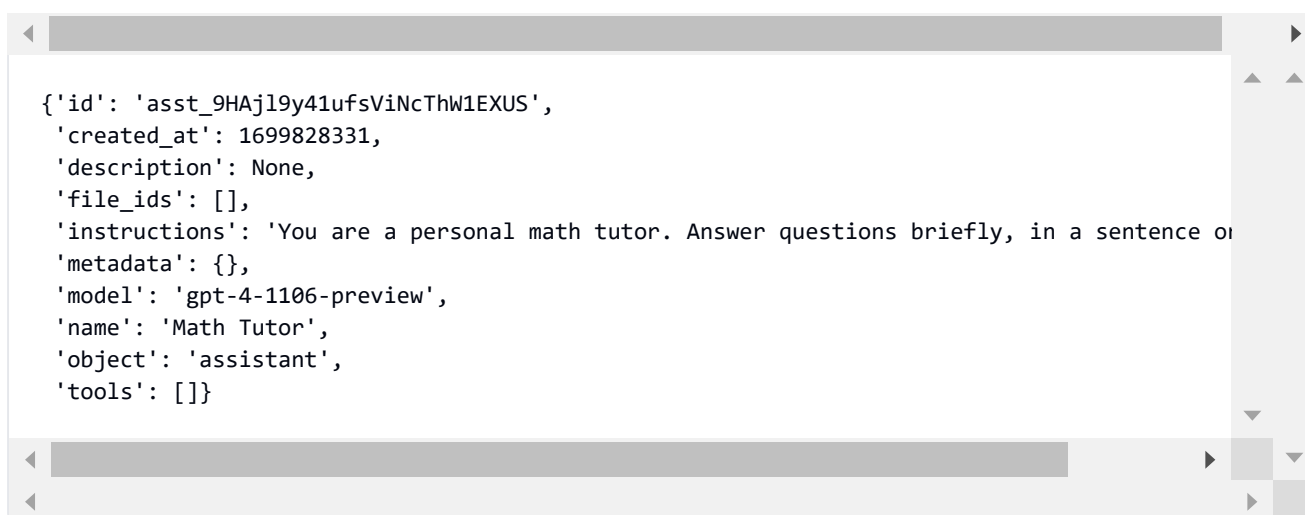
Name	Instructions	ID	Date Created
Math Tutor	You are a personal math tutor. Answer questions in a sentence or less.	asst_tN4AcVrQcfff6dRyvvtPhSz	Nov 8, 2023, 3:33 PM

You can also create Assistants directly through the Assistants API, like so:

```
from openai import OpenAI

client = OpenAI()

assistant = client.beta.assistants.create(
    name="Math Tutor",
    instructions="You are a personal math tutor. Answer questions briefly, in a sentence or less.",
    model="gpt-4-1106-preview",
)
show_json(assistant)
```



The screenshot shows a JSON response from the OpenAI Assistants API. The JSON object contains the following fields: 'id', 'created_at', 'description', 'file_ids', 'instructions', 'metadata', 'model', 'name', 'object', and 'tools'. The 'instructions' field is truncated in the image.

```
{
  'id': 'asst_9HAjl9y41ufsViNcThW1EXUS',
  'created_at': 1699828331,
  'description': None,
  'file_ids': [],
  'instructions': 'You are a personal math tutor. Answer questions briefly, in a sentence or',
  'metadata': {},
  'model': 'gpt-4-1106-preview',
  'name': 'Math Tutor',
  'object': 'assistant',
  'tools': []
}
```

Regardless of whether you create your Assistant through the Dashboard or with the API, you'll want to keep track of the Assistant ID. This is how you'll refer to your Assistant

throughout Threads and Runs.

Next, we'll create a new Thread and add a Message to it. This will hold the state of our conversation, so we don't have re-send the entire message history each time.

Threads

Create a new thread:

```
thread = client.beta.threads.create()  
show_json(thread)
```

```
{'id': 'thread_bw42vPoQtYBMQE84WubNcJXG',  
 'created_at': 1699828331,  
 'metadata': {},  
 'object': 'thread'}
```

Then add the Message to the thread:

```
message = client.beta.threads.messages.create(  
    thread_id=thread.id,  
    role="user",  
    content="I need to solve the equation `3x + 11 = 14`. Can you help me?",  
)  
show_json(message)
```

```
{'id': 'msg_IBiZDAWHhWPewxzN0EfTYNew',  
 'assistant_id': None,  
 'content': [{'text': {'annotations': [],  
                        'value': 'I need to solve the equation `3x + 11 = 14`. Can you help me?'},  
              'type': 'text'}],  
 'created_at': 1699828332,  
 'file_ids': [],  
 'metadata': {},  
 'object': 'thread.message',  
 'role': 'user',  
 'run_id': None,  
 'thread_id': 'thread_bw42vPoQtYBMQE84WubNcJXG'}
```

“Note Even though you're no longer sending the entire history each time, you will still be charged for the tokens of the entire conversation history with each Run.”

Runs

Notice how the Thread we created is **not** associated with the Assistant we created earlier! Threads exist independently from Assistants, which may be different from what you'd expect if you've used ChatGPT (where a thread is tied to a model/GPT).

To get a completion from an Assistant for a given Thread, we must create a Run. Creating a Run will indicate to an Assistant it should look at the messages in the Thread and take action: either by adding a single response, or using tools.

“Note Runs are a key difference between the Assistants API and Chat Completions API. While in Chat Completions the model will only ever respond with a single message, in the Assistants API a Run may result in an Assistant using one or multiple tools, and potentially adding multiple messages to the Thread.”

To get our Assistant to respond to the user, let's create the Run. As mentioned earlier, you must specify *both* the Assistant and the Thread.

```
run = client.beta.threads.runs.create(  
    thread_id=thread.id,  
    assistant_id=assistant.id,  
)  
show_json(run)
```

```
{'id': 'run_LA08RjouV3RemQ78UZXuyzv6',  
 'assistant_id': 'asst_9HAjl9y41ufsViNcThW1EXUS',  
 'cancelled_at': None,  
 'completed_at': None,  
 'created_at': 1699828332,  
 'expires_at': 1699828932,  
 'failed_at': None,  
 'file_ids': [],  
 'instructions': 'You are a personal math tutor. Answer questions briefly, in a sentence or',  
 'last_error': None,  
 'metadata': {},  
 'model': 'gpt-4-1106-preview',  
 'object': 'thread.run',  
 'required_action': None,  
 'started_at': None,  
 'status': 'queued',  
 'thread_id': 'thread_bw42vPoQtYBMQE84WubNcJXG',  
 'tools': []}
```

Unlike creating a completion in the Chat Completions API, **creating a Run is an asynchronous operation**. It will return immediately with the Run's metadata, which includes

a `status` that will initially be set to `queued`. The `status` will be updated as the Assistant performs operations (like using tools and adding messages).

To know when the Assistant has completed processing, we can poll the Run in a loop. (Support for streaming is coming soon!) While here we are only checking for a `queued` or `in_progress` status, in practice a Run may undergo a **variety of status changes** which you can choose to surface to the user. (These are called Steps, and will be covered later.)

```
import time

def wait_on_run(run, thread):
    while run.status == "queued" or run.status == "in_progress":
        run = client.beta.threads.runs.retrieve(
            thread_id=thread.id,
            run_id=run.id,
        )
        time.sleep(0.5)
    return run
```

```
run = wait_on_run(run, thread)
show_json(run)
```

```
{'id': 'run_LA08RjouV3RemQ78UZXuyzv6',
 'assistant_id': 'asst_9HAjl9y41ufsViNcThW1EXUS',
 'cancelled_at': None,
 'completed_at': 1699828333,
 'created_at': 1699828332,
 'expires_at': None,
 'failed_at': None,
 'file_ids': [],
 'instructions': 'You are a personal math tutor. Answer questions briefly, in a sentence or',
 'last_error': None,
 'metadata': {},
 'model': 'gpt-4-1106-preview',
 'object': 'thread.run',
 'required_action': None,
 'started_at': 1699828332,
 'status': 'completed',
 'thread_id': 'thread_bw42vPoQtYBMQE84WubNcJXG',
 'tools': []}
```

Messages

Now that the Run has completed, we can list the Messages in the Thread to see what got added by the Assistant.

```
messages = client.beta.threads.messages.list(thread_id=thread.id)
show_json(messages)
```



```
{'data': [{'id': 'msg_S0ZtKIWjyWtbIW9JNUocPdUS',
  'assistant_id': 'asst_9HAjl9y41ufsViNcThW1EXUS',
  'content': [{'text': {'annotations': [],
    'value': 'Yes. Subtract 11 from both sides to get `3x = 3`, then divide by 3 to find
    'type': 'text'}},
  'created_at': 1699828333,
  'file_ids': [],
  'metadata': {},
  'object': 'thread.message',
  'role': 'assistant',
  'run_id': 'run_LA08RjouV3RemQ78UZxuyzv6',
  'thread_id': 'thread_bw42vPoQtYBMQE84WubNcJXG'},
  {'id': 'msg_IBiZDAWHhWPewxzN0EfTYNew',
  'assistant_id': None,
  'content': [{'text': {'annotations': [],
    'value': 'I need to solve the equation `3x + 11 = 14`. Can you help me?'},
  'type': 'text'}},
  'created_at': 1699828332,
  'file_ids': [],
  'metadata': {},
  'object': 'thread.message',
  'role': 'user',
  'run_id': None,
  'thread_id': 'thread_bw42vPoQtYBMQE84WubNcJXG'}],
  'object': 'list',
  'first_id': 'msg_S0ZtKIWjyWtbIW9JNUocPdUS',
  'last_id': 'msg_IBiZDAWHhWPewxzN0EfTYNew',
  'has_more': False}
```



As you can see, Messages are ordered in reverse-chronological order – this was done so the most recent results are always on the first page (since results can be paginated). Do keep a look out for this, since this is the opposite order to messages in the Chat Completions API.

Let's ask our Assistant to explain the result a bit further!

```
# Create a message to append to our thread
message = client.beta.threads.messages.create(
    thread_id=thread.id, role="user", content="Could you explain this to me?"
)

# Execute our run
run = client.beta.threads.runs.create(
    thread_id=thread.id,
    assistant_id=assistant.id,
)

# Wait for completion
wait_on_run(run, thread)

# Retrieve all the messages added after our last user message
messages = client.beta.threads.messages.list(
```




```

        thread_id=thread.id, order="asc", after=message.id
    )
    show_json(messages)

```

```

{'data': [{'id': 'msg_9MAeOrGriHcImeQnAzvYyJbs',
  'assistant_id': 'asst_9HAjl9y41ufsViNcThW1EXUS',
  'content': [{'text': {'annotations': [],
    'value': 'Certainly. To solve for x in the equation `3x + 11 = 14`:\\n\\n1. Subtract 11:
    'type': 'text'}}],
  'created_at': 1699828335,
  'file_ids': [],
  'metadata': {},
  'object': 'thread.message',
  'role': 'assistant',
  'run_id': 'run_IFHfsubkJv7RSUbDZpNVs4PG',
  'thread_id': 'thread_bw42vPoQtYBMQE84WubNcJXG'}],
  'object': 'list',
  'first_id': 'msg_9MAeOrGriHcImeQnAzvYyJbs',
  'last_id': 'msg_9MAeOrGriHcImeQnAzvYyJbs',
  'has_more': False}

```

This may feel like a lot of steps to get a response back, especially for this simple example. However, you'll soon see how we can add very powerful functionality to our Assistant without changing much code at all!

Example

Let's take a look at how we could potentially put all of this together. Below is all the code you need to use an Assistant you've created.

Since we've already created our Math Assistant, I've saved its ID in `MATH_ASSISTANT_ID`. I then defined two functions:

- `submit_message` : create a Message on a Thread, then start (and return) a new Run
- `get_response` : returns the list of Messages in a Thread

```

from openai import OpenAI

MATH_ASSISTANT_ID = assistant.id # or a hard-coded ID like "asst-..."

client = OpenAI()

def submit_message(assistant_id, thread, user_message):
    client.beta.threads.messages.create(
        thread_id=thread.id, role="user", content=user_message
    )
    return client.beta.threads.runs.create(
        thread_id=thread.id,

```



```

        assistant_id=assistant_id,
    )

def get_response(thread):
    return client.beta.threads.messages.list(thread_id=thread.id, order="asc")

```

I've also defined a `create_thread_and_run` function that I can re-use (which is actually almost identical to the `client.beta.threads.create_and_run` compound function in our API 🤖). Finally, we can submit our mock user requests each to a new Thread.

Notice how all of these API calls are asynchronous operations; this means we actually get async behavior in our code without the use of async libraries! (e.g. `asyncio`)

```

def create_thread_and_run(user_input):
    thread = client.beta.threads.create()
    run = submit_message(MATH_ASSISTANT_ID, thread, user_input)
    return thread, run

# Emulating concurrent user requests
thread1, run1 = create_thread_and_run(
    "I need to solve the equation `3x + 11 = 14`. Can you help me?"
)
thread2, run2 = create_thread_and_run("Could you explain linear algebra to me?")
thread3, run3 = create_thread_and_run("I don't like math. What can I do?")

# Now all Runs are executing...

```



Once all Runs are going, we can wait on each and get the responses.

```

import time

# Pretty printing helper
def pretty_print(messages):
    print("# Messages")
    for m in messages:
        print(f"{m.role}: {m.content[0].text.value}")
    print()

# Waiting in a loop
def wait_on_run(run, thread):
    while run.status == "queued" or run.status == "in_progress":
        run = client.beta.threads.runs.retrieve(
            thread_id=thread.id,
            run_id=run.id,
        )
        time.sleep(0.5)
    return run

# Wait for Run 1
run1 = wait_on_run(run1, thread1)

```



```
pretty_print(get_response(thread1))

# Wait for Run 2
run2 = wait_on_run(run2, thread2)
pretty_print(get_response(thread2))

# Wait for Run 3
run3 = wait_on_run(run3, thread3)
pretty_print(get_response(thread3))

# Thank our assistant on Thread 3 🤖
run4 = submit_message(MATH_ASSISTANT_ID, thread3, "Thank you!")
run4 = wait_on_run(run4, thread3)
pretty_print(get_response(thread3))
```

```
# Messages
user: I need to solve the equation `3x + 11 = 14`. Can you help me?
assistant: Yes, subtract 11 from both sides to get `3x = 3`, then divide both sides by 3 to get `x = 1`.

# Messages
user: Could you explain linear algebra to me?
assistant: Linear algebra is the branch of mathematics that deals with vector spaces, linear transformations, and systems of linear equations.

# Messages
user: I don't like math. What can I do?
assistant: Try finding aspects of math that relate to your interests or daily life, and connect them to mathematical concepts.

# Messages
user: I don't like math. What can I do?
assistant: Try finding aspects of math that relate to your interests or daily life, and connect them to mathematical concepts.
user: Thank you!
assistant: You're welcome! If you have any more questions, feel free to ask.
```

Et voilà!

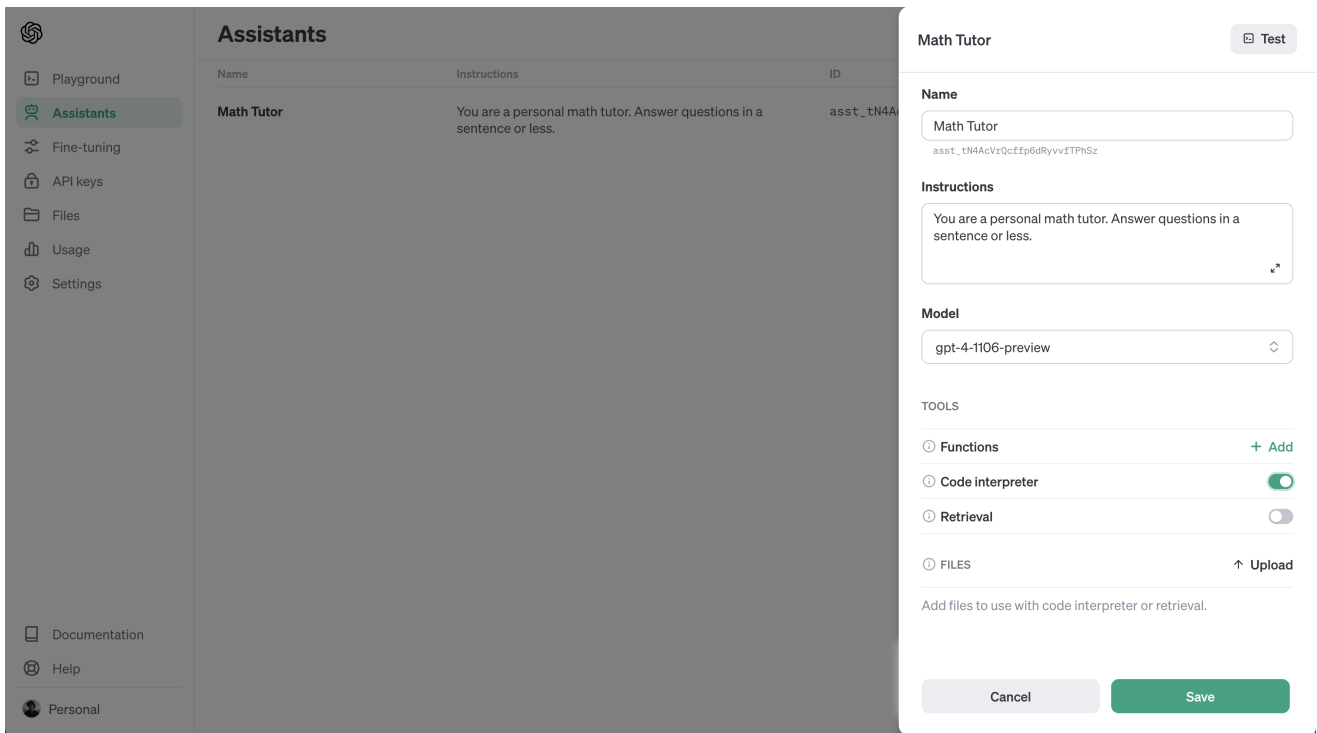
You may have noticed that this code is not actually specific to our math Assistant at all... this code will work for any new Assistant you create simply by changing the Assistant ID! That is the power of the Assistants API.

Tools

A key feature of the Assistants API is the ability to equip our Assistants with Tools, like Code Interpreter, Retrieval, and custom Functions. Let's take a look at each.

Code Interpreter

Let's equip our Math Tutor with the [Code Interpreter](#) tool, which we can do from the Dashboard...



...or the API, using the Assistant ID.

```
assistant = client.beta.assistants.update(
    MATH_ASSISTANT_ID,
    tools=[{"type": "code_interpreter"}],
)
show_json(assistant)
```

```
{'id': 'asst_9HAj19y41ufsViNcThW1EXUS',
 'created_at': 1699828331,
 'description': None,
 'file_ids': [],
 'instructions': 'You are a personal math tutor. Answer questions briefly, in a sentence or',
 'metadata': {},
 'model': 'gpt-4-1106-preview',
 'name': 'Math Tutor',
 'object': 'assistant',
 'tools': [{'type': 'code_interpreter'}]}
```

Now, let's ask the Assistant to use its new tool.

```
thread, run = create_thread_and_run(
    "Generate the first 20 fibonacci numbers with code."
)
run = wait_on_run(run, thread)
pretty_print(get_response(thread))
```

```
# Messages
user: Generate the first 20 fibonacci numbers with code.
assistant: The first 20 Fibonacci numbers are: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144
```

And that's it! The Assistant used Code Interpreter in the background, and gave us a final response.

For some use cases this may be enough – however, if we want more details on what precisely an Assistant is doing we can take a look at a Run's Steps.

Steps

A Run is composed of one or more Steps. Like a Run, each Step has a `status` that you can query. This is useful for surfacing the progress of a Step to a user (e.g. a spinner while the Assistant is writing code or performing retrieval).

```
run_steps = client.beta.threads.runs.steps.list(
    thread_id=thread.id, run_id=run.id, order="asc"
)
```

Let's take a look at each Step's `step_details`.

```
for step in run_steps.data:
    step_details = step.step_details
    print(json.dumps(show_json(step_details), indent=4))
```

```
{'tool_calls': [{'id': 'call_WMnQd63PtX8vZzTwaA6eWpBg',
  'code_interpreter': {'input': '# Python function to generate the first 20 Fibonacci numbers',
    'outputs': [{'logs': '[0,\n 1,\n 1,\n 2,\n 3,\n 5,\n 8,\n 13,\n 21,\n 34,\n 55,\n 89,\n 144]'}],
    'type': 'logs'}}],
  'type': 'code_interpreter'}],
  'type': 'tool_calls'}
```

```
null
```

```
{'message_creation': {'message_id': 'msg_z5931E5bvcD6BngeDFHDxzwM'},
  'type': 'message_creation'}
```

```
null
```

We can see the `step_details` for two Steps:

1. `tool_calls` (plural, since it could be more than one in a single Step)
2. `message_creation`

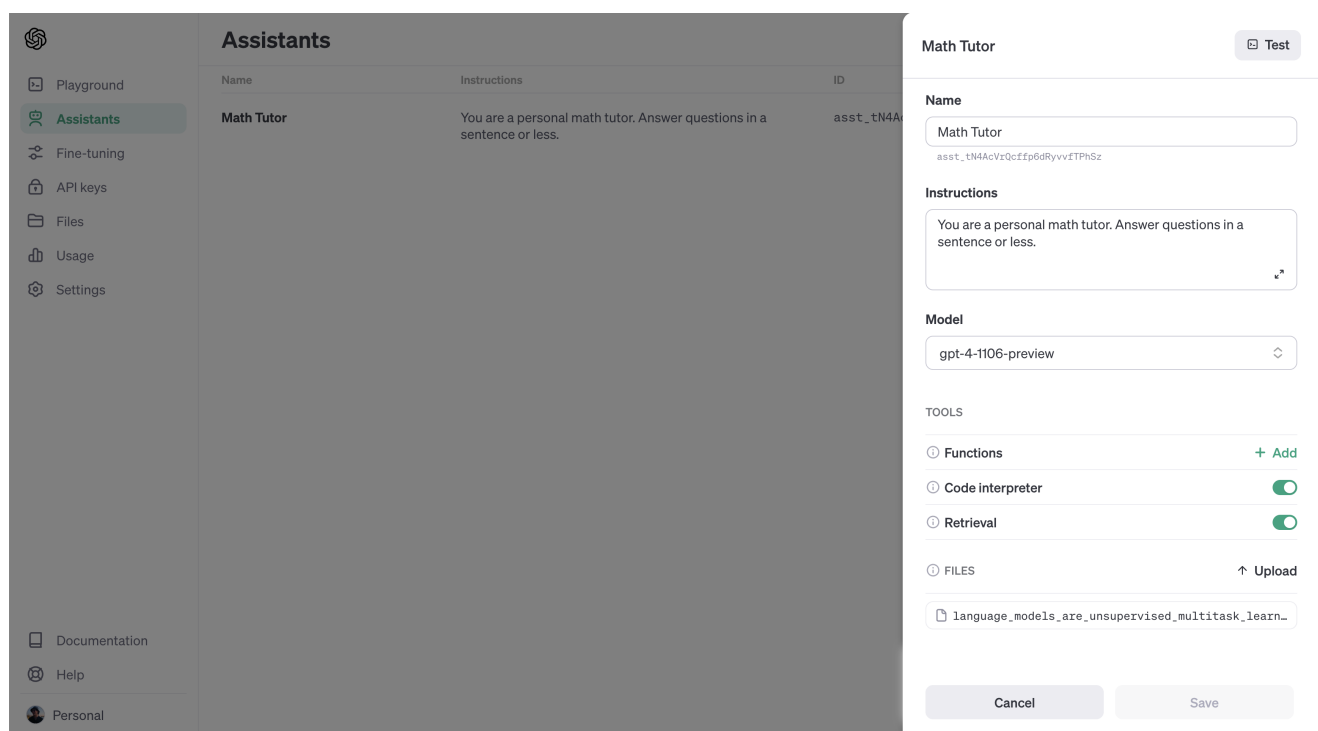
The first Step is a `tool_calls`, specifically using the `code_interpreter` which contains:

- `input`, which was the Python code generated before the tool was called, and
- `output`, which was the result of running the Code Interpreter.

The second Step is a `message_creation`, which contains the `message` that was added to the Thread to communicate the results to the user.

Retrieval

Another powerful tool in the Assistants API is **Retrieval**: the ability to upload files that the Assistant will use as a knowledge base when answering questions. This can also be enabled from the Dashboard or the API, where we can upload files we want to be used.



```
# Upload the file
file = client.files.create(
    file=open(
        "data/language_models_are_unsupervised_multitask_learners.pdf",
        "rb",
    ),
    purpose="assistants",
)
# Update Assistant
assistant = client.beta.assistants.update(
    MATH_ASSISTANT_ID,
    tools=[{"type": "code_interpreter"}, {"type": "retrieval"}],
    file_ids=[file.id],
)
show_json(assistant)
```

```
{'id': 'asst_9HAj19y41ufsViNcThw1EXUS',
 'created_at': 1699828331,
 'description': None,
 'file_ids': ['file-MdXcQI80dPp76wukWI4dpLwW'],
 'instructions': 'You are a personal math tutor. Answer questions briefly, in a sentence or two.',
 'metadata': {},
 'model': 'gpt-4-1106-preview',
 'name': 'Math Tutor',
 'object': 'assistant',
 'tools': [{'type': 'code_interpreter'}, {'type': 'retrieval'}]}
```

```
thread, run = create_thread_and_run(
    "What are some cool math concepts behind this ML paper pdf? Explain in two sentences."
)
run = wait_on_run(run, thread)
pretty_print(get_response(thread))
```

```
# Messages
user: What are some cool math concepts behind this ML paper pdf? Explain in two sentences.
assistant: I am unable to find specific sections referring to "cool math concepts" directly.
assistant: The paper discusses leveraging large language models as a framework for unsupervised learning.
```

"Note There are more intricacies in Retrieval, like Annotations, which may be covered in another cookbook."

Functions

As a final powerful tool for your Assistant, you can specify custom **Functions** (much like the **Function Calling** in the Chat Completions API). During a Run, the Assistant can then indicate it wants to call one or more functions you specified. You are then responsible for calling the Function, and providing the output back to the Assistant.

Let's take a look at an example by defining a `display_quiz()` Function for our Math Tutor.

This function will take a `title` and an array of `questions`, display the quiz, and get input from the user for each:

- `title`
- `questions`
 - `question_text`
 - `question_type : [MULTIPLE_CHOICE , FREE_RESPONSE]`
 - `choices : ["choice 1", "choice 2", ...]`

Unfortunately I don't know how to get user input within a Python Notebook, so I'll be mocking out responses with `get_mock_response...`. This is where you'd get the user's actual input.

```
def get_mock_response_from_user_multiple_choice():  
    return "a"  
  
def get_mock_response_from_user_free_response():  
    return "I don't know."  
  
def display_quiz(title, questions):  
    print("Quiz:", title)  
    print()  
    responses = []  
  
    for q in questions:  
        print(q["question_text"])  
        response = ""  
  
        # If multiple choice, print options  
        if q["question_type"] == "MULTIPLE_CHOICE":  
            for i, choice in enumerate(q["choices"]):  
                print(f"{i}. {choice}")  
            response = get_mock_response_from_user_multiple_choice()  
  
        # Otherwise, just get response  
        elif q["question_type"] == "FREE_RESPONSE":  
            response = get_mock_response_from_user_free_response()  
  
        responses.append(response)  
    print()
```




```
return responses
```

Here's what a sample quiz would look like:

```
responses = display_quiz(
    "Sample Quiz",
    [
        {"question_text": "What is your name?", "question_type": "FREE_RESPONSE"},
        {
            "question_text": "What is your favorite color?",
            "question_type": "MULTIPLE_CHOICE",
            "choices": ["Red", "Blue", "Green", "Yellow"],
        },
    ],
)
print("Responses:", responses)
```

Quiz: Sample Quiz

What is your name?

What is your favorite color?

- 0. Red
- 1. Blue
- 2. Green
- 3. Yellow

Responses: ["I don't know.", 'a']

Now, let's define the interface of this function in JSON format, so our Assistant can call it:

```
function_json = {
    "name": "display_quiz",
    "description": "Displays a quiz to the student, and returns the student's response. A single",
    "parameters": {
        "type": "object",
        "properties": {
            "title": {"type": "string"},
            "questions": {
                "type": "array",
                "description": "An array of questions, each with a title and potentially options",
                "items": {
                    "type": "object",
                    "properties": {
                        "question_text": {"type": "string"},
                        "question_type": {
                            "type": "string",
                            "enum": ["MULTIPLE_CHOICE", "FREE_RESPONSE"],
                        },
                        "choices": {"type": "array", "items": {"type": "string"}},
                    },
                    "required": ["question_text"],
                },
            },
        },
    },
}
```

```

    },
  },
  "required": ["title", "questions"],
},
}

```

Once again, let's update our Assistant either through the Dashboard or the API.

The screenshot shows the OpenAI Assistants Dashboard. On the left is a sidebar with navigation links: Playground, Assistants (selected), Fine-tuning, API keys, Files, Usage, and Settings. The main area is titled 'Assistants' and contains a table with columns 'Name', 'Instructions', and 'ID'. One assistant is listed: 'Math Tutor' with instructions 'You are a personal math tutor. Answer questions in a sentence or less.' and ID 'asst_tN4A...'. On the right, a 'Function' editor is open, showing a JSON definition for a quiz function. The function has parameters for 'name', 'parameters', 'type', 'properties', 'questions', 'items', 'question_text', 'question_type', 'enum', and 'choices'. The 'enum' includes 'MULTIPLE_CHOICE' and 'FREE_RESPONSE'. The 'choices' are defined as an array of strings.

"Note Pasting the function JSON into the Dashboard was a bit finicky due to indentation, etc. I just asked ChatGPT to format my function the same as one of the examples on the Dashboard 🤖."

```

assistant = client.beta.assistants.update(
    MATH_ASSISTANT_ID,
    tools=[
        {"type": "code_interpreter"},
        {"type": "retrieval"},
        {"type": "function", "function": function_json},
    ],
)
show_json(assistant)

```

```

{'id': 'asst_9HAj19y41ufsViNcThW1EXUS',
 'created_at': 1699828331,
 'description': None,

```

```
'file_ids': ['file-MdXcQI80dPp76wukWI4dpLwW'],
'instructions': 'You are a personal math tutor. Answer questions briefly, in a sentence or two.',
'metadata': {},
'model': 'gpt-4-1106-preview',
'name': 'Math Tutor',
'object': 'assistant',
'tools': [{ 'type': 'code_interpreter' },
{ 'type': 'retrieval' },
{ 'function': { 'name': 'display_quiz',
'parameters': { 'type': 'object',
'properties': { 'title': { 'type': 'string' },
'questions': { 'type': 'array',
'description': 'An array of questions, each with a title and potentially options (if multiple choice)',
'items': { 'type': 'object',
'properties': { 'question_text': { 'type': 'string' },
'question_type': { 'type': 'string',
'enum': ['MULTIPLE_CHOICE', 'FREE_RESPONSE'] },
'choices': { 'type': 'array', 'items': { 'type': 'string' } } },
'required': ['question_text'] } } },
'required': ['title', 'questions'] },
'description': "Displays a quiz to the student, and returns the student's response. A string of the student's response." } } } ] }
```

And now, we ask for a quiz.

```
thread, run = create_thread_and_run(
    "Make a quiz with 2 questions: One open ended, one multiple choice. Then, give me feedback for my answer."
)
run = wait_on_run(run, thread)
run.status
```

```
'requires_action'
```

Now, however, when we check the Run's status we see `requires_action` ! Let's take a closer look.

```
show_json(run)
```

```
{ 'id': 'run_98PGE3qGtHoawACLoytyRUBf',
'assistant_id': 'asst_9HAjl9y41ufsViNcThW1EXUS',
'cancelled_at': None,
'completed_at': None,
'created_at': 1699828370,
'expires_at': 1699828970,
'failed_at': None,
'file_ids': ['file-MdXcQI80dPp76wukWI4dpLwW'],
'instructions': 'You are a personal math tutor. Answer questions briefly, in a sentence or two.',
'last_error': None,
```

```

'metadata': {},
'model': 'gpt-4-1106-preview',
'object': 'thread.run',
'required_action': {'submit_tool_outputs': {'tool_calls': [{'id': 'call_Zf650sWT1wW4Uwbf5',
  'function': {'arguments': '{\n  "title": "Mathematics Quiz",\n  "questions": [\n    {\n      'name': 'display_quiz'},
      'type': 'function'}}]},
  'type': 'submit_tool_outputs'},
'started_at': 1699828370,
'status': 'requires_action',
'thread_id': 'thread_bICTESFvWoRdj000SzsosLCS',
'tools': [{ 'type': 'code_interpreter'},
  { 'type': 'retrieval'},
  { 'function': { 'name': 'display_quiz',
    'parameters': { 'type': 'object',
      'properties': { 'title': { 'type': 'string'},
        'questions': { 'type': 'array',
          'description': 'An array of questions, each with a title and potentially options (if multiple choice)',
          'items': { 'type': 'object',
            'properties': { 'question_text': { 'type': 'string'},

```

The `required_action` field indicates a Tool is waiting for us to run it and submit its output back to the Assistant. Specifically, the `display_quiz` function! Let's start by parsing the name and arguments.

"Note While in this case we know there is only one Tool call, in practice the Assistant may choose to call multiple tools."

```

# Extract single tool call
tool_call = run.required_action.submit_tool_outputs.tool_calls[0]
name = tool_call.function.name
arguments = json.loads(tool_call.function.arguments)

print("Function Name:", name)
print("Function Arguments:")
arguments

```

```

Function Name: display_quiz
Function Arguments:

```

```

{'title': 'Mathematics Quiz',
 'questions': [{'question_text': 'Explain why the square root of a negative number is not a real number.',
  'question_type': 'FREE_RESPONSE'},
  {'question_text': 'What is the value of an angle in a regular pentagon?',
  'choices': ['72 degrees', '90 degrees', '108 degrees', '120 degrees'],
  'question_type': 'MULTIPLE_CHOICE'}]}

```

Now let's actually call our `display_quiz` function with the arguments provided by the Assistant:

```
responses = display_quiz(arguments["title"], arguments["questions"])
print("Responses:", responses)
```

Quiz: Mathematics Quiz

Explain why the square root of a negative number is not a real number.

What is the value of an angle in a regular pentagon?

- 0. 72 degrees
- 1. 90 degrees
- 2. 108 degrees
- 3. 120 degrees

Responses: ["I don't know.", 'a']

Great! (Remember these responses are the one's we mocked earlier. In reality, we'd be getting input from the back from this function call.)

Now that we have our responses, let's submit them back to the Assistant. We'll need the `tool_call` ID, found in the `tool_call` we parsed out earlier. We'll also need to encode our list of responses into a `str`.

```
run = client.beta.threads.runs.submit_tool_outputs(
    thread_id=thread.id,
    run_id=run.id,
    tool_outputs=[
        {
            "tool_call_id": tool_call.id,
            "output": json.dumps(responses),
        }
    ],
)
show_json(run)
```

```
{'id': 'run_98PGE3qGtHoawACLOYtyRUBf',
 'assistant_id': 'asst_9HAj19y41ufsViNcThw1EXUS',
 'cancelled_at': None,
 'completed_at': None,
 'created_at': 1699828370,
 'expires_at': 1699828970,
 'failed_at': None,
 'file_ids': ['file-MdXcQI80dPp76wukWI4dpLwW'],
 'instructions': 'You are a personal math tutor. Answer questions briefly, in a sentence or',
 'last_error': None,
 'metadata': {},
 'model': 'gpt-4-1106-preview',
 'object': 'thread.run',
```

```
'required_action': None,
'started_at': 1699828370,
'status': 'queued',
'thread_id': 'thread_bICTESFvWoRdj000SzsosLCS',
'tools': [{ 'type': 'code_interpreter'},
{'type': 'retrieval'},
{'function': {'name': 'display_quiz',
'parameters': {'type': 'object',
'properties': {'title': {'type': 'string'},
'questions': {'type': 'array',
'description': 'An array of questions, each with a title and potentially options (i:
'items': {'type': 'object',
'properties': {'question_text': {'type': 'string'},
'question_type': {'type': 'string',
'enum': ['MULTIPLE_CHOICE', 'FREE_RESPONSE']},
'choices': {'type': 'array', 'items': {'type': 'string'}}}},
'required': ['question_text']}}}]}
```

We can now wait for the Run to complete once again, and check our Thread!

```
run = wait_on_run(run, thread)
pretty_print(get_response(thread))
```

Messages

user: Make a quiz with 2 questions: One open ended, one multiple choice. Then, give me feedback.
assistant: Thank you for attempting the quiz.

For the first question, it's important to know that the square root of a negative number is imaginary.
For the second question, the correct answer is "108 degrees." In a regular pentagon, which

Woohoo 🎉

Conclusion

We covered a lot of ground in this notebook, give yourself a high-five! Hopefully you should now have a strong foundation to build powerful, stateful experiences with tools like Code Interpreter, Retrieval, and Functions!

There's a few sections we didn't cover for the sake of brevity, so here's a few resources to explore further:

- **Annotations**: parsing file citations
- **Files**: Thread scoped vs Assistant scoped
- **Parallel Function Calls**: calling multiple tools in a single Step

- Multi-Assistant Thread Runs: single Thread with Messages from multiple Assistants
- Streaming: coming soon!

Now go off and build something amazing!