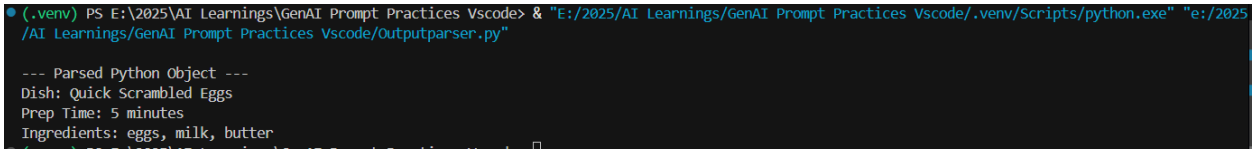
Structured Output using json 👍

Code:

```
from pydantic import BaseModel, Field
from google import genai
import os
from google.genai import types
from dotenv import load_dotenv
import json

load_dotenv()

# 2. Check the environment for the key and pass it to the Client.
# The os.environ dictionary is updated by load_dotenv().
API_KEY = os.environ.get("GEMINI_API_KEY")

if not API_KEY:
    raise ValueError("GEMINI_API_KEY not found in environment variables.")

# 3. Instantiate the client, passing the key explicitly.
client = genai.Client(api_key=API_KEY)

class Recipe(BaseModel):
    """A structured recipe object."""
    dish_name: str = Field(description="The name of the dish.")
    prep_time_minutes: int = Field(description="The preparation time in
minutes.")
    ingredients: list[str] = Field(description="A list of required
ingredients.")
    is_vegetarian: bool = Field(description="True if the recipe is
strictly vegetarian.")

# Example: A prompt that requires a recipe
RECIPE_PROMPT = (
    "Generate a simple, quick recipe for a breakfast dish using only 3
ingredients. "
    "**Respond with a single, clean JSON object that strictly adheres to
the provided schema.** "
```

```
    "DO NOT include any introductory or explanatory text outside of the
JSON block."
)
response = client.models.generate_content(
    model='gemini-2.5-flash',
    contents=RECIPE_PROMPT,
    config=types.GenerateContentConfig(
        # CRITICAL: Pass the Pydantic class to response_schema
        response_schema=Recipe,
        response_mime_type="application/json"

        # Optional: Set the response mime type to JSON for robustness
        # response_mime_type="application/json"
    )
)

# NOTE: The response.text will now contain a RAW JSON string.
raw_json_string = response.text

try:
    # 1. Use the Pydantic model to parse and validate the JSON string
    structured_recipe = Recipe.model_validate_json(raw_json_string)

    # 2. Access the data as a normal Python object (no messy
dictionaries!)
    print("\n--- Parsed Python Object ---")
    print(f"Dish: {structured_recipe.dish_name}")
    print(f"Prep Time: {structured_recipe.prep_time_minutes} minutes")
    print(f"Ingredients: {', '.join(structured_recipe.ingredients)}")

except json.JSONDecodeError:
    print("Error: The model failed to return valid JSON.")
except ValueError as e:
    print(f"Error: The JSON format was invalid for the schema. Details:
{e}")
```

Output:



Reference(Google Gemini):

Structured Output (JSON)Forcing the LLM's response into a predictable format (like a Python dictionary or JSON object) using schema definitions.Crucial for building reliable applications where you need to parse the output, not just read it.

we now focus on this

guide me how to do it and help me to become expert in this

That's an excellent next step! **Structured Output (JSON)** is fundamental to moving from simply chatting with an LLM to **building reliable applications** that use AI.

You achieve structured output by defining a **schema** (a blueprint) for the desired JSON, and then you tell the model to strictly follow that schema when generating its response.

Here is your guide to mastering Structured Output using the `google-genai` SDK.

---

1. Core Concept: Defining the Schema

To get a structured response, you first need a Python data structure to represent the format. The `google-genai` SDK uses the standard Python library **Pydantic** to define these schemas.

A. Define the Pydantic Schema

In your `Code.py` file, you need to import the necessary classes and define what your JSON output should look like.

Python
```
from pydantic import BaseModel, Field
from google import genai
import os # Assuming you still use dotenv for setup

# --- Your existing setup code (load_dotenv, client instantiation) goes here ---
# client = genai.Client(api_key=API_KEY)
```

```
# ----------------------------------------------------------------------

# 1. DEFINE THE OUTPUT SCHEMA using Pydantic
class Recipe(BaseModel):
    """A structured recipe object."""
    dish_name: str = Field(description="The name of the dish.")
    prep_time_minutes: int = Field(description="The preparation time in minutes.")
    ingredients: list[str] = Field(description="A list of required ingredients.")
    is_vegetarian: bool = Field(description="True if the recipe is strictly vegetarian.")

# Example: A prompt that requires a recipe
RECIPE_PROMPT = "Generate a simple, quick recipe for a breakfast dish using only 3 ingredients."
```

B. Why Pydantic is Crucial

- **Enforcement:** Pydantic models define the **required types** (`str`, `int`, `list[str]`, `bool`). If the LLM generates a response that doesn't match the schema (e.g., puts text where an integer belongs), the API will usually catch the error, or your Python code will fail immediately and reliably, preventing runtime bugs.
- **Descriptions:** The `Field(description=...)` text is sent to the LLM. This is vital because it acts as **additional guidance** to ensure the model knows *what* kind of data to put in each field.

---

2. Implementation: Forcing JSON Output

You use the `response_schema` parameter within the `GenerateContentConfig` object to instruct the model to return its output as a JSON object that matches your `Recipe` schema.

Python

```
from google.genai import types

# Use the client and RECIPE_PROMPT defined above
response = client.models.generate_content(
    model='gemini-2.5-flash',
    contents=RECIPE_PROMPT,
    config=types.GenerateContentConfig(
        # CRITICAL: Pass the Pydantic class to response_schema
        response_schema=Recipe,
```

```
        # Optional: Set the response mime type to JSON for robustness
        # response_mime_type="application/json"
    )
)

# NOTE: The response.text will now contain a RAW JSON string.
raw_json_string = response.text
print("--- Raw JSON Output from LLM ---")
print(raw_json_string)

# The next step is to parse this raw JSON string back into a usable Python object.
```

---

3. Post-Processing: Using the Structured Data

Once you have the raw JSON string from the model, you use the Pydantic model again to reliably parse and validate it.

A. Parsing the JSON

The Pydantic class you defined (`Recipe`) has a built-in method (`model_validate_json()`) to convert the raw JSON string back into a Python object you can work with.

Python
```
import json
# Assuming 'raw_json_string' holds the model's response text

try:
    # 1. Use the Pydantic model to parse and validate the JSON string
    structured_recipe = Recipe.model_validate_json(raw_json_string)

    # 2. Access the data as a normal Python object (no messy dictionaries!)
    print("\n--- Parsed Python Object ---")
    print(f"Dish: {structured_recipe.dish_name}")
    print(f"Prep Time: {structured_recipe.prep_time_minutes} minutes")
    print(f"Ingredients: {', '.join(structured_recipe.ingredients)}")

except json.JSONDecodeError:
    print("Error: The model failed to return valid JSON.")
except ValueError as e:
    print(f"Error: The JSON format was invalid for the schema. Details: {e}")
```

Becoming an Expert in Structured Output

To master this technique, focus on these advanced concepts:

1. **Schema Complexity:** Practice creating nested schemas (e.g., a `Menu` object that contains a `list[Recipe]` object). This is crucial for handling complex, multi-object responses.
2. **Enforcing Constraints:** Learn how to use Pydantic validators (like minimum/maximum length, default values, and custom validators) to impose strict business logic on the data the LLM generates.
3. **Error Handling:** Always wrap your parsing code (`model_validate_json()`) in a `try...except` block. A model can sometimes still fail to produce perfect JSON, and robust applications must handle these failures gracefully (e.g., by logging the failure and retrying the request).
4. **Tool Use:** Once comfortable with structured output, the next logical step is **Function Calling**, which uses Pydantic schemas to define the *input arguments* for functions the LLM can decide to call.
    - **Goal:** The model returns a structured JSON telling you which function to call and what arguments to pass, based on the user's request. This is the foundation of building a true AI Agent.

Error: The JSON format was invalid for the schema. Details: 1 validation error for Recipe

Invalid JSON: expected value at line 1 column 1 [type=json_invalid, input_value='Okay, here\'s a super si...of the 3 ingredients!).', input_type=str]

For further information visit https://errors.pydantic.dev/2.11/v/json_invalid

This error is excellent! It means you've progressed beyond simple installation failures and are now debugging the core logic of your AI application. 🎉

The error: **`ValueError: Invalid JSON: expected value at line 1 column 1`** means the model did **not** return a clean JSON string; it returned plain English text instead.

Specifically, the model started its response with the text: `"Okay, here's a super si...of the 3 ingredients!)."` and then likely stopped or provided invalid JSON.

The Problem: The Model Ignored the Schema

When you use structured output, the LLM sometimes fails to follow the instructions and adds conversational filler (like "Okay, here's the answer...") before or after the required JSON block. Your Python code receives this text and, when it tries to run

`Recipe.model_validate_json(raw_json_string)`, it fails immediately because the string doesn't start with a JSON object (`{`).

Solution: Force Strict JSON Mode

To fix this, you must explicitly tell the model to return **only** a valid JSON object and nothing else. You do this in two ways:

1. Update the Configuration (Required)

Ensure your `client.models.generate_content` call is using the `response_mime_type` parameter to force the model into strict JSON output.

Python
```
from google.genai import types

response = client.models.generate_content(
    model='gemini-2.5-flash',
    contents=RECIPE_PROMPT,
    config=types.GenerateContentConfig(
        # 1. CRITICAL: Pass the Pydantic class to define the structure
        response_schema=Recipe,

        # 2. CRITICAL: Force the model to output a clean JSON string
        response_mime_type="application/json"
    )
)
```

2. Improve the Prompt (Highly Recommended)

While the configuration in Step 1 is essential, adding clear constraints in the prompt acts as extra enforcement for the model's behavior.

Modify your `RECIPE_PROMPT` to include a clear instruction:

Python
```
RECIPE_PROMPT = (
    "Generate a simple, quick recipe for a breakfast dish using only 3 ingredients. "
    "**Respond with a single, clean JSON object that strictly adheres to the provided schema.** "
    "DO NOT include any introductory or explanatory text outside of the JSON block."
)
```

By combining the **strict configuration** (`response_mime_type="application/json"`) and the **explicit prompt instruction**, you drastically increase the chance of the model returning a perfect JSON string that your `model_validate_json()` call can process successfully.

If the error persists after these changes, try simplifying the prompt further and ensuring your Pydantic class definition is correct.