Learning Objectives

This notebook teaches how to solve the problem of unstructured or unreliable data from LLMs by using the Pydantic library to define, validate, and parse data into a reliable structure.

After completing this notebook, you will be able to:

- Explain the challenges of using raw LLM-generated JSON, such as extraneous text and incorrect data types that cause parsing errors.
- Define a Pydantic model using `BaseModel` to enforce a specific data structure with typed fields (e.g., `str` , `int` , `List`).
- Instantiate a Pydantic model from a dictionary of data and access its validated attributes.
- Interpret Pydantic `ValidationError` messages to pinpoint data quality issues.
- Generate a machine-readable JSON Schema from a Pydantic model.
- Explain "constrained generation" and use a Pydantic-generated schema within an LLM API call to force the model to return perfectly structured, valid JSON.
- Implement a complete workflow to convert unstructured text into a validated Pydantic object using an LLM API.

Enforcing Data Schemas with Pydantic and LLMs

Working with Structured Data in Programming

Programming operates on structured data principles. Modern software systems communicate through standardized formats, store information in organized schemas, and process data according to defined patterns. This structured approach enables different systems to interact seamlessly and ensures data integrity across applications.

The Challenge of Unstructured Output

Consider a practical scenario in healthcare. When a doctor dictates clinical notes, the natural language output might appear as:

Doctor's Note: "47-year-old man with history of high blood pressure and type 2 diabetes presents with chest pain for 3 days. Pain radiates to left arm and jaw, worsens with exertion, improves with rest. Associated with shortness of breath, nausea, sweating. Exam: elevated blood pressure, fast heart rate, low oxygen. Lungs clear, no murmurs. Tests: elevated troponin, ECG shows ST depression in leads II, III, and aVF. Impression: Likely acute coronary syndrome. Consider unstable angina or non-ST elevation heart attack."

While this narrative format serves human readers well, databases and applications require structured data. Electronic health records, billing systems, and clinical decision support tools need information organized in a predictable, machine-readable format. For instance, using the following dictionary:

```
{
    "age": 47,
    "sex": "male",
    "history": ["high blood pressure", "type 2 diabetes"],
    "main_symptom": "chest pain for 3 days, radiating to arm and jaw, worse with exertion, better with
rest",
    "associated_symptoms": ["shortness of breath", "nausea", "sweating"],
    "exam_findings": ["elevated blood pressure", "fast heart rate", "low oxygen", "lungs clear", "no
murmurs"],
    "tests": ["elevated troponin", "ECG with ST depression in leads II, III, aVF"],
    "impression": "acute coronary syndrome (possible unstable angina or non-ST elevation heart
attack)"
}
```

This structured format enables seamless integration with laboratory systems, insurance providers, and clinical databases. Each piece of information is stored in a specific field with a defined data type (e.g., integer age, list of associated_symptoms, etc.), making automated processing possible.

The standards are usually so critical that even minor variations in structure can break system compatibility. Consider this alternative format:

```
{
    "patient_age": 47,
    "sex": "M",
    "history": {"high blood pressure", "type 2 diabetes"},
    "main_symptom": ["chest pain for 3 days, radiating to arm and jaw, worse with exertion, better
with rest"],
    "symptoms": {"shortness of breath", "nausea", "sweating"},
    "exam_findings": ["elevated blood pressure", "fast heart rate", "low oxygen", "lungs clear", "no
murmurs"],
    "tests": ["elevated troponin", "ECG with ST depression in leads II, III, aVF"],
    "impression": "acute coronary syndrome (possible unstable angina or non-ST elevation heart
attack)"
}
```

The differences may seem trivial:

The differences may seem trivial:

- The field `age` becomes `patient_age`
- The value for `sex` changes from "male" to "M"
- `symptoms` uses a set structure instead of a list
- etc...

These inconsistencies prevent automated systems from processing the data correctly. A program expecting `age` will fail when encountering `patient_age` . Code designed to iterate through a list of symptoms will crash when receiving a set. Systems expecting the full word "male" cannot process the abbreviation "M" without additional mapping logic.

Converting Text to Structured Data with LLMs

Large Language Models offer a promising solution for converting unstructured text into structured formats. The straightforward approach involves providing explicit instructions to convert text into a desired format. For instance:

```
Convert the following text to a JSON object with this exact structure:
{
    "age": <patient's age as an integer>,
    "sex": <patient's sex>,
    "history": [list of medical history items],
    "main_symptom": "description of primary symptom",
    "associated_symptoms": [list of accompanying symptoms],
    "exam_findings": [list of examination results],
    "tests": [list of test results],
    "impression": "clinical assessment"
}
```

While LLMs often produce reasonable results, several issues arise:

Issue 1: Extraneous Text

The LLM might return:

```
Here is the JSON output as requested:
{
    "age": 47,
    "sex": "male",
    ...
}
```

This response includes explanatory text that prevents direct JSON parsing. When a Python program attempts to process this output:

In [31]:
```
import json
LLM_output = """
{
    "age": 47,
    "sex": "male",
    "history": ["high blood pressure", "type 2 diabetes"],
    "main_symptom": "chest pain for 3 days, radiating to arm and jaw, worse with exertion, better wi
    "associated_symptoms": ["shortness of breath", "nausea", "sweating"],
    "exam_findings": ["elevated blood pressure", "fast heart rate", "low oxygen", "lungs clear", "no
    "tests": ["elevated troponin", "ECG with ST depression in leads II, III, aVF"],
    "impression": "acute coronary syndrome (possible unstable angina or non-ST elevation heart attac
}
"""
patient_record = json.loads(LLM_output)
print(patient_record)
```

```
{'age': 47, 'sex': 'male', 'history': ['high blood pressure', 'type 2 diabetes'], 'main_symptom': 'chest pain for 3 days, radiating to arm and jaw, worse with exertion, better with rest', 'associated_symptoms': ['shortness of breath', 'nausea', 'sweating'], 'exam_findings': ['elevated blood pressure', 'fast heart rate', 'low oxygen', 'lungs clear', 'no murmurs'], 'tests': ['elevated troponin', 'ECG with ST depression in leads II, III, aVF'], 'impression': 'acute coronary syndrome (possible unstable angina or non-ST elevation heart attack)'}
```

In [34]:
```
patient_record['age'] < 45
```

Out[34]:False

In [36]:
```
# the following fails due to the text returned by the model.

import json
LLM_output = """
Here is the JSON output as requested:
{
    "age": 47,
    "sex": "male",
    "history": ["high blood pressure", "type 2 diabetes"],
    "main_symptom": "chest pain for 3 days, radiating to arm and jaw, worse with exertion, better wi
```

```
        "associated_symptoms": ["shortness of breath", "nausea", "sweating"],
        "exam_findings": ["elevated blood pressure", "fast heart rate", "low oxygen", "lungs clear", "no
        "tests": ["elevated troponin", "ECG with ST depression in leads II, III, aVF"],
        "impression": "acute coronary syndrome (possible unstable angina or non-ST elevation heart attac
    }"""
    patient_record = json.loads(LLM_output)   # This fails!
```

```
---------------------------------------------------------------------------
JSONDecodeError                           Traceback (most recent call last)
Cell In[36], line 14
      1 import json
      2 LLM_output = """
      3 Here is the JSON output as requested:
      4 {
   (...)
     12     "impression": "acute coronary syndrome (possible unstable angina or non-ST elevation heart attack)"
     13 }"""
---> 14 patient_record = json.loads(LLM_output)

File ~/miniconda3/envs/py3.12/lib/python3.12/json/__init__.py:346, in loads(s, cls, object_hook, parse_float, parse_int, parse_constant, obje
ct_pairs_hook, **kw)
    341     s = s.decode(detect_encoding(s), 'surrogatepass')
    343 if (cls is None and object_hook is None and
    344         parse_int is None and parse_float is None and
    345         parse_constant is None and object_pairs_hook is None and not kw):
--> 346     return _default_decoder.decode(s)
    347 if cls is None:
    348     cls = JSONDecoder

File ~/miniconda3/envs/py3.12/lib/python3.12/json/decoder.py:338, in JSONDecoder.decode(self, s, _w)
    333 def decode(self, s, _w=WHITESPACE.match):
    334     """Return the Python representation of ``s`` (a ``str`` instance
    335     containing a JSON document).
    336 
    337     """
--> 338     obj, end = self.raw_decode(s, idx=_w(s, 0).end())
    339     end = _w(s, end).end()
    340     if end != len(s):

File ~/miniconda3/envs/py3.12/lib/python3.12/json/decoder.py:356, in JSONDecoder.raw_decode(self, s, idx)
    354     obj, end = self.scan_once(s, idx)
    355 except StopIteration as err:
--> 356     raise JSONDecodeError("Expecting value", s, err.value) from None
    357 return obj, end

JSONDecodeError: Expecting value: line 2 column 1 (char 1)
```

In the above `json.loads()` function expects pure JSON and raises an error when encountering the explanatory text.

While the returned format may be converted to JSON without any errors, this does not mean that the data is valid.

For exmaple, while the following data setructure is valid json,

In [40]:
```
LLM_output = """
    {
        "age": "47 years old",
        "sex": "male",
        "history": ["high blood pressure", "type 2 diabetes"],
        "main_symptom": "chest pain for 3 days, radiating to arm and jaw, worse with exertion, better wi
        "associated_symptoms": ["shortness of breath", "nausea", "sweating"],
        "exam_findings": ["elevated blood pressure", "fast heart rate", "low oxygen", "lungs clear", "no
        "tests": ["elevated troponin", "ECG with ST depression in leads II, III, aVF"],
        "impression": "acute coronary syndrome (possible unstable angina or non-ST elevation heart attac

    }
    """
    patient_record = json.loads(LLM_output)
    patient_record
```

```
Out[40]:{'age': '47 years old',
         'sex': 'male',
         'history': ['high blood pressure', 'type 2 diabetes'],
         'main_symptom': 'chest pain for 3 days, radiating to arm and jaw, worse with exertion, better with rest',
         'associated_symptoms': ['shortness of breath', 'nausea', 'sweating'],
         'exam_findings': ['elevated blood pressure',
          'fast heart rate',
          'low oxygen',
          'lungs clear',
          'no murmurs'],
         'tests': ['elevated troponin',
          'ECG with ST depression in leads II, III, aVF'],
         'impression': 'acute coronary syndrome (possible unstable angina or non-ST elevation heart attack)'}
```

The comparison fails because `age` contains the string "47 years old" rather than the integer 47. Python cannot compare a string to a number directly, resulting in a TypeError.

```
In [41]:patient_record['age'] > 45  # This fails!
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[41], line 1
----> 1 patient_record['age'] > 45

TypeError: '>' not supported between instances of 'str' and 'int'
```

The Root Problem: Ambiguity

The instruction "put the patient's age here" allows multiple valid interpretations:

- `47` (integer)
- `"47"` (string)
- `"47 years old"` (descriptive string)
- `"47 y/o"` (abbreviated string)

Each interpretation seems reasonable to the LLM, but only one works with downstream systems expecting an integer value. The Solution for this is using an explicit library to vaidate data. So, rather than trusting LLMs to interpret requirements correctly, we need a mechanism to:

1. Define exact data structures with explicit types, i.e., being explicit and precise about the expected output
2. Validate LLM outputs against these structures
3. Provide clear feedback when outputs don't match requirements
4. Enable iterative refinement until outputs conform

Pydantic excels at these tasks. As Python's most widely-used data validation library (https://pydantic-docs.helpmanual.io/), Pydantic creates data models that define both structure and data types for expected outputs.

Defining Pydantic Models

Consider a student information system requiring:

- `first_name` : string value
- `last_name` : string value
- `age` : integer value
- `major` : restricted to "Computer Science", "Mathematics", or "Nursing"
- `email` : properly formatted email address

Valid data would look like:

```
{
    'first_name': 'John',
    'last_name': 'Doe',
    'age': 27,
    'major': 'Computer Science',
    'email': 'john.doe@hawaii.edu'
}
```

Invalid examples include:

```
# Invalid: abbreviated major
{
    'first_name': 'John',
    'last_name': 'Doe',
    'age': 27,
    'major': 'Comp Sci.',   # Not in allowed values
    'email': 'john.doe@hawaii.edu'
}

# Invalid: wrong major and malformed email
{
    'first_name': 'John',
    'last_name': 'Doe',
    'age': 27,
    'major': 'Business',   # Not an allowed major
    'email': 'john.doe@'   # Invalid email format
}
```

Installing and Using Pydantic

Install Pydantic using pip:

```
pip install pydantic
```

Then define the data model:

In [ ]:
```
from pydantic import BaseModel, EmailStr
from typing import Literal

class StudentInfo(BaseModel):
    first_name: str
    last_name: str
    age: int
    major: Literal["Computer Science", "Mathematics", "Nursing"]
```

```
        email: EmailStr
```
The syntax used here will be covered in more detail in the object-oriented programming section, but suffice it to say that the `StudentInfo` class defined here is a `BaseModel` , Pydantic's foundational class. By being a subclass of `BaseModel` , `StudentInfo` inherits—or automatically acquires—built-in validation capabilities.

Each line in within the class defines a field with its expected type:

- `first_name: str` - Accepts any text string for the student's first name
- `last_name: str` - Accepts any text string for the student's last name
- `age: int` - Requires an integer value (whole number) for age
- `major: Literal[...]` - Restricts input to exactly one of the three specified strings. Any other value, even slight variations like "computer science" (lowercase) or "Math" (abbreviated), will be rejected
- `email: EmailStr` - Validates that the input follows standard email formatting rules (contains @ symbol, has valid domain structure, etc.)

When data is provided to this model, Pydantic automatically:

1. Checks that all required fields are present
2. Validates that each field's value matches its specified type
3. Converts compatible values when possible (e.g., the string "27" to integer 27)
4. Raises clear error messages for any validation failures

For instance, we can pass the data directly to the the class and see it "construct" and object of type info if the data is valid or return an error if the data is no vlaid. For exmaple the following passes muster

Working with Pydantic Objects

When constructing a Pydantic object, the data dictionary must be unpacked using the `**` operator:

```
data = {
    'first_name': 'John',
    'last_name': 'Doe',
    'age': 27,
    'major': 'Computer Science',
    'email': 'john.doe@hawaii.edu'
}
student_data = StudentInfo(**data)
```
The `**data` syntax deserves special attention. This double-asterisk operator simply means that we need to pass the data contained in the dictionary, rather that the dictionary itself, i.e., without it, Python would pass the entire dictionary as a single argument, causing an error. The "unpacking" operation transforms `{'first_name': 'John', 'last_name': 'Doe', ...}` into the equivalent of writing `StudentInfo(first_name='John', last_name='Doe', ...)` . This concept will be explored in greater detail in later chapters on Python functions and arguments.

The constructed object displays as:

```
StudentInfo(first_name='John', last_name='Doe', age=27, major='Computer Science',
email='john.doe@hawaii.edu')
```

In [ ]:
```
# this yields an error
StudentInfo({"first_name": 'John', "last_name": 'Doe', "age": 27, "major": 'Computer Science', "email"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[129], line 2
      1 # this yields an error
----> 2 StudentInfo({"first_name": 'John', "last_name": 'Doe', "age": 27, "major": 'Computer Science', "email": 'john.doe@hawaii.edu'})

TypeError: BaseModel.__init__() takes 1 positional argument but 2 were given
```

In [ ]:
```
# while this unpacking operation works
StudentInfo(**{"first_name": 'John', "last_name": 'Doe', "age": 27, "major": 'Computer Science', "emai
```

Out[ ]:StudentInfo(first_name='John', last_name='Doe', age=27, major='Computer Science', email='john.doe@hawaii.edu')

In [131]:
```
# the unpacking operation is equivalent to
  StudentInfo(first_name='John', last_name='Doe', age=27, major='Computer Science', email='john.doe@ha
```

Out[131]:StudentInfo(first_name='John', last_name='Doe', age=27, major='Computer Science', email='john.doe@hawaii.edu')

Let's assign the new object to a new varaible

In [ ]:
```
student_data = StudentInfo(first_name='John', last_name='Doe', age=27, major='Computer Science', email
```
Pydantic models provide multiple ways to access stored data. The dot notation offers direct field access:

In [133]:
```
student_data.first_name
```

Out[133]:'John'

In [ ]:
```
student_data.email
```

Out[ ]:'john.doe@hawaii.edu'

To retrieve all data as a dictionary, use the `model_dump()` method:

```
In [135]:student_data.model_dump()
```

```
Out[135]:{'first_name': 'John',
          'last_name': 'Doe',
          'age': 27,
          'major': 'Computer Science',
          'email': 'john.doe@hawaii.edu'}
```

Validation in Action

Pydantic enforces validation rules strictly. Invalid data triggers clear, informative error messages:

```
In [137]:data = {
             'first_name': 'John',
             'last_name': 'Doe',
             'age': 27,
             'major': 'Comp Sci.',   # Not in allowed values
             'email': 'john.doe@hawaii.edu'
         }

         student_data = StudentInfo(**data)
```

```
---------------------------------------------------------------------------
ValidationError                           Traceback (most recent call last)
Cell In[137], line 9
      1 data = {
      2     'first_name': 'John',
      3     'last_name': 'Doe',
   (...)
      6     'email': 'john.doe@hawaii.edu'
      7 }
----> 9 student_data = StudentInfo(**data)

File ~/miniconda3/envs/py3.12/lib/python3.12/site-packages/pydantic/main.py:253, in BaseModel.__init__(self, **data)
    251 # `__tracebackhide__` tells pytest and some other tools to omit this function from tracebacks
    252 __tracebackhide__ = True
--> 253 validated_self = self.__pydantic_validator__.validate_python(data, self_instance=self)
    254 if self is not validated_self:
    255     warnings.warn(
    256         'A custom validator is returning a value other than `self`.\n'
    257         "Returning anything other than `self` from a top level model validator isn't supported when validating via `__init__`.\n"
    258         'See the `model_validator` docs (https://docs.pydantic.dev/latest/concepts/validators/#model-validators) for more details.',
    259         stacklevel=2,
    260     )

ValidationError: 1 validation error for StudentInfo
major
  Input should be 'Computer Science', 'Mathematics' or 'Nursing' [type=literal_error, input_value='Comp Sci.', input_type=str]
    For further information visit https://errors.pydantic.dev/2.11/v/literal_error
```

This produces a `ValidationError`:

```
ValidationError: 1 validation error for StudentInfo
major
  Input should be 'Computer Science', 'Mathematics' or 'Nursing'
  [type=literal_error, input_value='Comp Sci.', input_type=str]
```

The error message precisely identifies the problematic field (`major`) and explains why the value was rejected. Similarly, invalid email formats trigger validation errors:

```
In [138]:data = {
             'first_name': 'John',
             'last_name': 'Doe',
             'age': 27,
             'major': 'Marine Sciences',   # Not an allowed major
             'email': 'john.doe@'   # Invalid email format
         }
         student_data = StudentInfo(**data)
```

```
---------------------------------------------------------------------------
ValidationError                           Traceback (most recent call last)
Cell In[138], line 8
      1 data = {
      2     'first_name': 'John',
      3     'last_name': 'Doe',
   (...)
      6     'email': 'john.doe@'  # Invalid email format
      7 }
----> 8 student_data = StudentInfo(**data)

File ~/miniconda3/envs/py3.12/lib/python3.12/site-packages/pydantic/main.py:253, in BaseModel.__init__(self, **data)
    251 # `__tracebackhide__` tells pytest and some other tools to omit this function from tracebacks
    252 __tracebackhide__ = True
--> 253 validated_self = self.__pydantic_validator__.validate_python(data, self_instance=self)
    254 if self is not validated_self:
    255     warnings.warn(
    256         'A custom validator is returning a value other than `self`.\n'
    257         "Returning anything other than `self` from a top level model validator isn't supported when validating via `__init__`.\n"
    258         'See the `model_validator` docs (https://docs.pydantic.dev/latest/concepts/validators/#model-validators) for more details.',
    259         stacklevel=2,
    260     )

ValidationError: 2 validation errors for StudentInfo
major
  Input should be 'Computer Science', 'Mathematics' or 'Nursing' [type=literal_error, input_value='Marine Sciences', input_type=str]
    For further information visit https://errors.pydantic.dev/2.11/v/literal_error
email
  value is not a valid email address: There must be something after the @-sign. [type=value_error, input_value='john.doe@', input_type=str]
```

Advanced Validation Capabilities

Pydantic extends far beyond basic type checking. The library includes specialized validators for common data types (dates, URLs, IP addresses, credit card numbers) and supports custom validation logic. Developers can implement constraints such as:

- Age ranges (e.g., between 18 and 120)
- Domain restrictions for email addresses (e.g., must end with "@hawaii.edu")
- Conditional requirements (e.g., certain fields required only when others have specific values)
- Complex business rules specific to the application domain

Machine-Readable Schemas

One of Pydantic's most powerful features for LLM integration is its ability to generate JSON schemas—standardized descriptions of data structures that machines can interpret. The `model_json_schema()` method produces this representation:

In [139]:
```
StudentInfo.model_json_schema()
```

Out[139]:
```
{'properties': {'first_name': {'title': 'First Name', 'type': 'string'},
  'last_name': {'title': 'Last Name', 'type': 'string'},
  'age': {'title': 'Age', 'type': 'integer'},
  'major': {'enum': ['Computer Science', 'Mathematics', 'Nursing'],
   'title': 'Major',
   'type': 'string'},
  'email': {'format': 'email', 'title': 'Email', 'type': 'string'}},
 'required': ['first_name', 'last_name', 'age', 'major', 'email'],
 'title': 'StudentInfo',
 'type': 'object'}
```

While this format appears technical to human readers, it provides precise instructions that LLMs understand perfectly. The schema specifies:

- The model has a a bunch of properties (e.g., first_name, last_name, etc.)
- Each field's data type (`type`)
- Which fields are mandatory (`required`)
- Allowed values for restricted fields (`enum`)
- Special formatting requirements (`format`)

Integrating Pydantic with LLMs

The schema can be sent directly to an LLM API along with the data to be processed. Consider this practical example:

In [146]:
```
import requests

url = "https://api.openai.com/v1/chat/completions"
headers = {
    "Content-Type": "application/json",
    "Authorization": "Bearer YOUR_API_KEY_HERE"
}

data = {
    "model": "gpt-4o",
    "messages": [
        {
            "role": "system",
            "content": "You are a helpful assistant."
        },
        {
            "role": "user",
            "content": """
Consider the following text:

Jason Doe is 27 years old. He is majoring in Computer Science
and his email alias is jason27. The email domain is hawaii.edu.

Please convert it to JSON object following this schema:

{'properties': {'first_name': {'title': 'First Name', 'type': 'string'},
  'last_name': {'title': 'Last Name', 'type': 'string'},
  'age': {'title': 'Age', 'type': 'integer'},
  'major': {'enum': ['Computer Science', 'Mathematics', 'Nursing'],
   'title': 'Major',
   'type': 'string'},
  'email': {'format': 'email', 'title': 'Email', 'type': 'string'}},
 'required': ['first_name', 'last_name', 'age', 'major', 'email'],
 'title': 'StudentInfo',
 'type': 'object'}

"""
        }
    ]
}

response = requests.post(url, headers=headers, json=data)
resp_data = response.json()
resp_data
```

Out[146]:
```
{'id': 'chatcmpl-CCvfDLBfdA7iR88xr8Wx09NAtz16M',
 'object': 'chat.completion',
 'created': 1757197631,
 'model': 'gpt-4o-2024-08-06',
 'choices': [{'index': 0,
   'message': {'role': 'assistant',
    'content': 'Based on the schema provided and the information in the text, the JSON object would look like this:\n\n```json\n{\n  "first_name": "Jason",\n  "last_name": "Doe",\n  "age": 27,\n  "major": "Computer Science",\n  "email": "jason27@hawaii.edu"\n}\n```',
    'refusal': None,
    'annotations': []},
   'logprobs': None,
   'finish_reason': 'stop'}],
 'usage': {'prompt_tokens': 211,
  'completion_tokens': 71,
  'total_tokens': 282,
  'prompt_tokens_details': {'cached_tokens': 0, 'audio_tokens': 0},
  'completion_tokens_details': {'reasoning_tokens': 0,
   'audio_tokens': 0,
   'accepted_prediction_tokens': 0,
   'rejected_prediction_tokens': 0}},
 'service_tier': 'default',
 'system_fingerprint': 'fp_f33640a400'}
```

In [147]:
```
print(resp_data['choices'][0]['message']['content'])
```

~~~
Based on the schema provided and the information in the text, the JSON object would look like this:

```json
{
  "first_name": "Jason",
  "last_name": "Doe",
  "age": 27,
  "major": "Computer Science",
  "email": "jason27@hawaii.edu"
}
```
~~~

Notice how the model correctly:

- Extracted "Jason" as the first name from the full name "Jason Doe"
- Converted the age to an integer
- Matched "majoring in Computer Science" to the exact enum value
- Assembled the email from separate components (alias + domain)

However, despite understanding the schema, the LLM still embeds the JSON within explanatory text. This additional text prevents direct parsing and requires string manipulation to extract the JSON portion—a fragile approach prone to errors.

Constrained Generation: The Complete Solution

Modern LLM APIs support "constrained generation," forcing outputs to conform exactly to specified schemas without any extraneous text. This feature "guarantees" that responses can be parsed directly into Pydantic models.

The implementation requires adding a `response_format` parameter to the API request:

In [149]:
```
import requests
import json

my_student_url_schema = StudentInfo.model_json_schema()
my_student_url_schema["additionalProperties"] = False  # Prevent extra fields

url = "https://api.openai.com/v1/chat/completions"
headers = {
    "Content-Type": "application/json",
    "Authorization": "Bearer YOUR_API_KEY"
}

data = {
    "model": "gpt-4o",
    "messages": [
        {
            "role": "system",
            "content": "You are a helpful assistant."
        },
        {
            "role": "user",
            "content": """
Consider the following text:

Jason Doe is 27 years old. He is majoring in Computer Science
and his email alias is jason27. The email domain is hawaii.edu.
"""
        }
    ],
    "response_format": {
        "type": "json_schema",
        "json_schema": {
            "name": "student_info_extraction",
            "strict": True,  # Enforces strict schema adherence
            "schema": my_student_url_schema
        }
    }
}

response = requests.post(url, headers=headers, json=data)
response
```

Out[149]:
```
<Response [200]>
```

In [150]:
```
resp_data = response.json()
print(resp_data['choices'][0]['message']['content'])
```

```
{"first_name":"Jason","last_name":"Doe","age":27,"major":"Computer Science","email":"jason27@hawaii.edu"}
```

The `response_format` parameter instructs the LLM to:

- Output only valid JSON (`type: "json_schema"`)
- Follow the provided schema exactly (`schema: schema`)
- Reject any output that doesn't conform (`strict: True`)

The constrained response contains pure JSON, but formatted as a string:

```
'{"first_name":"Jason","last_name":"Doe","age":27,"major":"Computer
Science","email":"jason27@hawaii.edu"}'
```

{"first_name":"Jason","last_name":"Doe","age":27,"major":"Computer Science","email":"jason27@hawaii.edu"}

Complete Integration Workflow

With constrained generation, the entire process becomes seamless:

```
# 1. Get the LLM response
json_str = resp['choices'][0]['message']['content']
print(json_str)
# Output: {"first_name":"Jason","last_name":"Doe","age":27,"major":"Computer
Science","email":"jason27@hawaii.edu"}

# 2. Parse the JSON string
data = json.loads(json_str)
# Result: Python dictionary with properly typed values

# 3. Create the Pydantic model
student_data = StudentInfo(**data)
# Result: Validated StudentInfo object

# 4. Access the data with confidence
print(student_data.age)   # 27 (as integer)
print(student_data.email)  # jason27@hawaii.edu (validated email)
```

This workflow guarantees:

- No parsing errors from extraneous text
- All data conforms to specified types
- Invalid data is caught immediately
- Downstream systems receive exactly the structure they expect

Practical Exercise

Here you will convert unstructured text into validated data structures in a usecase that mirror a real-world scenario.

Create a Pydantic model for medical notes and use it to structure doctor's observations through an LLM API. Consider the following doctor's note.

"47-year-old man with history of high blood pressure and type 2 diabetes presents with chest pain for 3 days. Pain radiates to left arm and jaw, worsens with exertion, improves with rest. Associated with shortness of breath, nausea, sweating. Exam: elevated blood pressure, fast heart rate, low oxygen. Lungs clear, no murmurs. Tests: elevated troponin, ECG shows ST depression in leads II, III, and aVF. Impression: Likely acute coronary syndrome. Consider unstable angina or non-ST elevation heart attack."

Step 1: Define the Pydantic Model

Create a `MedicalNote` class that captures all the essential information from clinical notes. Your model should include:

- Patient demographics (age, sex as two separate fields)
- Medical history (list of conditions)
- Primary symptom description
- Associated list of symptoms
- Examination findings
- Test results
- Clinical impression

Refer to the Pydantic documentation at https://docs.pydantic.dev/latest/concepts/fields/ for available field types if needed.

In [ ]:
```
from pydantic import BaseModel
from typing import List, Literal

# YOUR CODE HERE: Define the MedicalNote class
class MedicalNote(BaseModel):
    # Define all necessary fields with appropriate types
    pass
```

Step 2: Generate the Schema

Extract the JSON schema from your model:

In [ ]:
```
# YOUR CODE HERE: Get and display the schema
my_medical_note_schema["additionalProperties"] = False = # Complete this line to add the schema.

print(my_medical_note_schema)
```

Step 3: Send to LLM API

Complete the API request to convert the doctor's note into structured data:

In [ ]:
```
import requests
import json

# YOUR CODE HERE: Complete the API request
url = "https://api.openai.com/v1/chat/completions"
headers = {
    "Content-Type": "application/json",
    "Authorization": "Bearer YOUR_API_KEY"
}

doctor_note = """
47-year-old man with history of high blood pressure and type 2 diabetes
presents with chest pain for 3 days. Pain radiates to left arm and jaw,
worsens with exertion, improves with rest. Associated with shortness of
breath, nausea, sweating. Exam: elevated blood pressure, fast heart rate,
low oxygen. Lungs clear, no murmurs. Tests: elevated troponin, ECG shows
ST depression in leads II, III, and aVF. Impression: Likely acute coronary
syndrome. Consider unstable angina or non-ST elevation heart attack.
"""

data = {
    # YOUR CODE HERE: Complete the request structure
    # Include model, messages, and response_format
}

response = requests.post(url, headers=headers, json=data)
```

Step 4: Validate and Use the Result

Parse the response and create a validated `MedicalNote` object:

In [ ]:
```
# YOUR CODE HERE: Parse response and create validated object
json_response = response.json()
medical_data_str = # Extract the content
medical_data_dict = # Parse the JSON string
medical_note = # Create the MedicalNote object
```

In [ ]:
```
# Test your implementation
print(f"Patient age: {medical_note.age}")
print(f"Number of symptoms: {len(medical_note.associated_symptoms)}")
print(f"Clinical impression: {medical_note.impression}")
```