7

AI and Advanced Data Transformations

Building on Chapter 6, where we harnessed AI to address data quality challenges, we now turn our focus to advanced data transformations. These transformations are essential in real-world data engineering, enabling us to manage complex data scenarios with precision.

Solving these complex data transformations traditionally requires a wide range of expertise, constant context switching, and familiarity with numerous tools, libraries, and requirements. This can be daunting and inefficient. AI offers a "one size fits all" solution through a conversational interface, simplifying the process by consolidating these diverse needs into a single, adaptable tool.

In this chapter, we continue to leverage AI, not as a code generator, but as a versatile tool—our multi-spanner in the data engineering toolkit. AI aids us in navigating the intricacies of data transformations, from handling nested structures to normalizing diverse datasets. While AI offers innovative solutions, it is crucial to remain aware of its limitations and ensure human oversight where necessary.

Join us as we explore the critical role of advanced transformations and how AI can enhance these processes, addressing common pitfalls and maximizing efficiency.

7.1 Complex Text Processing with Regular Expressions

Regular expressions (regex) are essential tools in data engineering for extracting structured data from unstructured text. As datasets grow, structural and formatting issues become common challenges that data engineers face. This section explores traditional regex techniques and AI-driven methods to automate and enhance regex pattern generation. We will focus on processing log files, a typical task in real-world data pipelines, to extract meaningful information from seemingly chaotic data entries.

Listing 7.1: Python - Complex Text Processing with Regular Expressions

```
import re  #A

# Example log entries  #B
logs = [  #C
    "ERROR 2025-06-09 12:34:56 Server failed to respond",  #D
    "INFO 2025-06-09 12:35:56 User logged in",  #E
    "WARNING 2025-06-09 12:36:56 Disk space low"  #F
]

# Multi-part pattern with capture groups  #G
pattern = r"(ERROR|INFO|WARNING)\s(\d{4}-\d{2}-\d{2})\s(\d{2}:\d{2}:\d{2})"  #H

# Extract log type, date, and time from each log entry  #I
for log in logs:  #J
    match = re.search(pattern, log)  #K
    if match:  #L
        log_type, date, time = match.groups()  #M
        print(f"Type: {log_type}, Date: {date}, Time: {time}")  #N
```

#A Import required libraries for regex operations.
#B–#F Define example log entries to simulate real-world scenarios.
#G–#H Create a regex pattern to capture log type, date, and time components.
#I–#N Iterate over log entries to extract and print each component using regex.

This regex extracts the date and time from each log entry using capture groups. This approach is efficient for structured data extraction but requires precise pattern definitions, which can become complex as data variability increases.

Listing 7.2: Python - Complex Text Processing with Regular Expressions

```
import openai  #A
import os  #B
from dotenv import load_dotenv  #C
from pydantic import BaseModel  #D
from typing import Optional  #E

# Load API key from .env file  #F
load_dotenv()  #G
openai.api_key = os.getenv("OPENAI_API_KEY")  #H

# Define the data model for extracted output  #I
class LogExtraction(BaseModel):  #J
    log_type: Optional[str]  #J.1
    date: Optional[str]  #K
    time: Optional[str]  #L

# Example log entries  #M
logs = [  #N
    "ERROR 2025-06-09 12:34:56 Server failed to respond",  #O
    "INFO 2025-06-09 12:35:56 User logged in",  #P
    "WARNING 2025-06-09 12:36:56 Disk space low"  #Q
]
```

```
# Prompt for AI to extract log_type, date, and time  #R
row_prompts = [  #S
    "You are a data extraction assistant. Extract the following from the log entry:\n"
    "- log_type: The type of log message (e.g., ERROR, INFO, WARNING)\n"
    "- date: Extract the date in YYYY-MM-DD format\n"
    "- time: Extract the time in HH:MM:SS format\n"
    "Return the result as a JSON object matching the LogExtraction structure."
    for log in logs  #T
]

# Process each log entry  #U
for log, prompt in zip(logs, row_prompts):  #V
    try:  #W
        # Make the API call  #X
        completion = openai.beta.chat.completions.parse(  #Y
            model="gpt-4o",  #Z
            messages=[  #AA
                {"role": "system", "content": prompt},  #AB
                {"role": "user", "content": log}  #AC
            ],  #AD
            response_format=LogExtraction  #AE
        )

        extracted = completion.choices[0].message.parsed.dict()  #AF
        print(extracted)  #AG

    except Exception as e:  #AH
        print(f"Error processing log entry: {e}")  #AI
```

#A–#E Import required libraries for environment loading, API access, and data modeling.
#F–#H Load the OpenAI API key securely using environment variables.
#I–#L Define a Pydantic model to structure the extracted fields: log type, date, and time.
#M–#Q Provide example log entries simulating different log levels and timestamps.
#R–#T Create individual prompts for each log entry instructing the AI to extract structured data.
#U–#AI Iterate through each log and prompt pair, send them to the OpenAI API, and print the extracted results or handle errors gracefully.

The traditional regex approach involves defining a precise pattern to match specific text formats. For example, the pattern r"(ERROR|INFO|WARNING)\s(\d{4}-\d{2}-\d{2})\s(\d{2}:\d{2}:\d{2})" is used to extract dates and times from log entries. This approach requires a deep understanding of regex syntax and the ability to craft patterns that accurately capture the desired data.

In contrast, the AI-driven approach leverages conversational prompts to instruct the AI on what to extract, allowing for more flexibility and adaptability to diverse data formats. This method simplifies the extraction process by using natural language instructions.

Pitfalls of the AI Approach

AI can enhance regex pattern generation, but it may also produce incorrect patterns due to hallucinations. It's crucial to validate AI-generated solutions and understand their limitations to ensure accuracy and reliability in data processing tasks. As rules and conditions pile up, maintaining accuracy and consistency becomes challenging, necessitating robust validation mechanisms.

Try it now

Modify Listing 7.2 to also extract the log message content following the timestamp. For example, in the log entry: ERROR 2025-06-09 12:34:56 Server failed to respond

You've already extracted the log_type, date, and time. Your new goal is to capture the message portion—Server failed to respond—and include it in the structured output as a new field (e.g., message).

Update the LogExtraction model to include the message field, modify the AI prompt to instruct it to extract this new component and rerun the code to verify the new output includes all four values: log_type, date, time, and message.

7.2 Handling Hierarchical and Nested Data Structures

Handling hierarchical and nested data structures is a common challenge in data engineering. These structures, often represented in formats like JSON or XML, can be complex and difficult to manage. From a technical perspective, efficiently processing these data types is crucial for data transformation and storage. From a business perspective, understanding and manipulating these structures can lead to more insightful data analysis and decision-making.

To illustrate these approaches, we will use a JSON dataset representing a library with nested details about books, authors, and genres. This dataset will help us demonstrate how traditional recursive functions and AI models can be applied to extract and transform nested data.


```
{
  "library": {
    "name": "City Library",
    "location": "Downtown",
    "books": [
      {
        "title": "Python Programming",
        "author": {
          "first_name": "John",
          "last_name": "Doe"
        },
        "genres": ["Programming", "Technology"],
        "published_year": 2020
      },
      {
        "title": "Data Science 101",
        "author": {
          "first_name": "Jane",
          "last_name": "Smith"
        },
        "genres": ["Data Science", "AI"],
        "published_year": 2019
      }
    ]
  }
}
```


Figure 7.1: Nested JSON data set containing information about books in a library.

Listing 7.3: Python - Handling Hierarchical and Nested Data Structures

```
import pandas as pd  #A

# Function to convert nested JSON to a DataFrame  #B
def json_to_dataframe(json_data):  #C
    """Convert nested JSON to a pandas DataFrame."""  #D
    records = []  #E
    for book in json_data['library']['books']:  #F
        record = {  #G
            'Library Name': json_data['library']['name'],  #H
            'Location': json_data['library']['location'],  #I
            'Title': book['title'],  #J
            'Author': f"{book['author']['first_name']} {book['author']['last_name']}",  #K
            'Genres': ', '.join(book['genres']),  #L
            'Published Year': book['published_year']  #M
        }
        records.append(record)  #N
    return pd.DataFrame(records)  #O

df = json_to_dataframe(json_data) #P
display(df) #Q
```

#A Import the pandas library for data manipulation and analysis.
#B–#D Define a function to transform nested JSON into a flat pandas DataFrame.
#E–#F Initialize a list to collect records and loop through each book entry in the nested JSON.
#G–#M Extract relevant fields from both the top-level and nested structures and format them into a flat dictionary.
#N Append each structured record to the list of records.
#O Convert the list of dictionaries into a pandas DataFrame and return it.
#P–#Q Call the transformation function and display the resulting DataFrame.

While the traditional Python approach in listing 7.3 looks clean and efficient, it comes with some hidden complexity. To flatten and extract the nested data correctly, you need to know the exact structure of the JSON, how to traverse it, how to concatenate nested fields like author, and how to transform arrays like genres into something readable. That means writing more code, handling edge cases manually, and knowing a fair bit about data wrangling in Python.

Listing 7.4: AI - Handling Hierarchical and Nested Data Structures

```
import openai  #A
import os  #B
from dotenv import load_dotenv  #C
from pydantic import BaseModel  #D
import pandas as pd  #E

load_dotenv()  #F
openai.api_key = os.getenv("OPENAI_API_KEY")  #G

class LibraryBook(BaseModel):  #H
    library_name: str  #I
    location: str  #J
    title: str  #K
    author: str  #L
    genres: str  #M
    published_year: int  #N

system_prompt = f"Extract data to match this class:\n{LibraryBook.schema_json(indent=2)}"
#O

structured_data = []  #P
```

```
for book in json_data["library"]["books"]:  #Q
    payload = {  #R
        "library_name": json_data["library"]["name"],  #S
        "location": json_data["library"]["location"],  #T
        **book  #U
    }

    try:  #V
        completion = openai.beta.chat.completions.parse(  #W
            model="gpt-4o",  #X
            messages=[  #Y
                {"role": "system", "content": system_prompt},  #Z
                {"role": "user", "content": f"{payload}"}  #AA
            ],
            response_format=LibraryBook  #AB
        )
        structured_data.append(completion.choices[0].message.parsed.dict())  #AC
    except Exception as e:  #AD
        print(f"Error: {e}")  #AE

df = pd.DataFrame(structured_data)  #AF
display(df)  #AG
```

#A–#E Import required libraries for OpenAI access, environment loading, data modeling, and tabular display.
#F–#G Load the OpenAI API key from a local .env file to securely authenticate the request.
#H–#N Define the target data structure (LibraryBook) using Pydantic, which will serve as the AI response format.
#O Create a dynamic prompt by extracting the class schema using Pydantic's built-in JSON formatter.
#P Initialize an empty list to hold the structured outputs returned by the AI.
#Q–#U Loop through each book entry and dynamically attach the shared context (`library_name`, `location`) without hardcoding.
#V–#W Use OpenAI's structured output API to generate a response from the AI based on the prompt and combined payload.
#X–#Y Set up the conversation using system and user messages, defining intent and data input.
#Z–#AA Pass the schema definition and single enriched book entry as the full input.
#AB Specify the expected response format using the LibraryBook class.
#AC–#AE Parse and append the AI's structured response, handling any errors that occur during the process.
#AF–#AG Convert the list of structured dictionaries into a DataFrame and display the result.

In listing 7.4, you describe the data structure you want using a simple data class, and the model figures out how to match it, even from a nested structure. The only "hardcoding" involved is making sure that shared attributes (like the library name and location) are included in the payload. Everything else is handled by the model, making the solution more extensible, especially as data structures become more complex. And best of all, when your schema changes, you don't have to rewrite your extraction logic, just update the class definition.

Try it now

Modify the JSON by adding one or more books with different structures—for example, an extra genre, a missing field, or a more complex author format. Update the code and run it through ChatGPT to see how the AI handles the changes.

Then, extend the LibraryBook class to include a new static attribute like library_type. Update the JSON accordingly and verify whether your current harness still works with minimal changes.

Finally, try adapting the code to a new JSON structure (such as a list of movies) by defining a new data class and prompt. See how much of the original code you can reuse.

7.3 String Normalization and Entity Resolution

In the world of data engineering, string normalization isn't just a cleanup task, it's how governance rules get enforced. Want every incoming email to follow a company-wide format? That's string normalization. Need to parse and extract domains or validate that an email is business-related, not personal? That's normalization too.

This section shows how normalization plays a crucial role in one of the most common real-world use cases: entity resolution. Imagine a support ticket comes in with a messy email address. Your system needs to decide, does this customer already exist in our system, or is this someone new? Getting that answer right requires normalized inputs and intelligent matching logic.

Here's the scenario: we've got a new email from jonny_smith@acmeco.com. It's got issues; nicknames, domain variation, no perfect match. We'll start by walking through a traditional Python approach to normalize and compare it against existing CRM entries:

```
customers = [
    {"name": "John Smith", "email": "john.smith@acme.com"},
    {"name": "Jane Smythe", "email": "jane.smythe@alpha.io"},
    {"name": "Jonathan Smith", "email": "jonathan@acme.com"},
    {"name": "Johnny S.", "email": "johnny@acmeco.com"}
]
incoming_email = "jonny_smith@acmeco.com"
```

Figure 7.2: Example existing customer records and incoming email of unknown customer.

We'll normalize the email, extract relevant parts, and then use fuzzy matching to identify possible matches. After that, we'll explore how AI can do the same task in a more flexible, conversational way, with a lot less code.

Listing 7.5: Python - String Normalization and Entity Resolution

```
import re  #A
from rapidfuzz import fuzz, process  #B

# Normalize the email by removing special characters and lowercasing  #C
def normalize_email(email):  #D
```

```
    prefix = email.split('@')[0]  #E
    normalized = re.sub(r'\W+', '', prefix.lower())  #F
    return normalized  #G

# Normalize the incoming email and customer emails  #H
normalized_incoming = normalize_email(incoming_email)  #I
customer_candidates = [  #J
    {
        "original": customer,  #K
        "normalized": normalize_email(customer["email"])  #L
    }
    for customer in customers  #M
]

# Score similarity using fuzzy matching  #N
matches = [  #O
    {
        "name": c["original"]["name"],  #P
        "email": c["original"]["email"],  #Q
        "score": fuzz.ratio(normalized_incoming, c["normalized"])  #R
    }
    for c in customer_candidates  #S
]
# Sort and print top match  #T
top_match = sorted(matches, key=lambda m: m["score"], reverse=True)[0]  #U
print(f"Best match: {top_match['name']} ({top_match['email']}) - Score:
{top_match['score']}")  #V
```

#A–#B Import libraries for regex and fuzzy string matching.
#C–#G Define a function to normalize email prefixes by removing special characters and lowercasing.
#H–#M Normalize the incoming email and all CRM customer email prefixes.
#N–#S Score each record's similarity to the incoming email using fuzzy string matching.
#T–#V Sort the matches by similarity score and print the top result.

This approach successfully identified the most likely match as John Smith, returning a score of 80.0. While the logic executes quickly and produces accurate results, it comes with a fair amount of overhead. We had to write a normalization function, construct intermediate candidate and match objects, and manually choose a similarity metric. More importantly, this method assumes we already know what kind of inconsistencies to expect and that we can handle them in advance.

The upcoming AI-based solution removes much of that burden, providing a more flexible, conversational way to resolve entities, even when the variations are unexpected or more nuanced.

Listing 7.6: AI - String Normalization and Entity Resolution

```
import openai  #A
import os  #B
from dotenv import load_dotenv  #C
from pydantic import BaseModel  #D

load_dotenv()  #E
openai.api_key = os.getenv("OPENAI_API_KEY")  #F

# Define a structured output model for entity resolution  #G
class EmailMatch(BaseModel):  #H
    name: str  #I
    email: str  #J
    confidence: float  #K
    reasoning: str  #L
```

```
# Define the system prompt with resolution instruction  #M
system_prompt = """
You are an entity resolution assistant. A new customer email has arrived:
'jonny_smith@acmeco.com'.
Compare it to the list of known customers. Identify the best match based on email
similarity and name inference.
Provide the closest match with a confidence score (0 to 1) and explain your reasoning.
""".strip()  #N

# Format the full input payload  #O
user_message = {
    "incoming_email": "jonny_smith@acmeco.com",
    "customers": customers
}  #P

# Call the OpenAI API with schema enforcement  #Q
completion = openai.beta.chat.completions.parse(  #R
    model="gpt-4o",  #S
    messages=[  #T
        {"role": "system", "content": system_prompt},  #U
        {"role": "user", "content": f"{user_message}"}  #V
    ],
    response_format=EmailMatch  #W
)

# Extract and display the structured response  #X
match = completion.choices[0].message.parsed.dict()  #Y

# Format and Print  #Z
print(f"Best match: {match['name']} ({match['email']})")
print(f"Confidence: {match['confidence']:.2f}")
print("Reasoning:")
print(match['reasoning'])
```

#A–#C Import libraries for environment setup and OpenAI access.
#D–#L Define the structured response format using EmailMatch, a Pydantic model.
#M–#N Compose a system prompt that explains the matching task to the AI.
#O–#P Provide the incoming email and previously defined customer_data as the input.
#Q–#W Send the request to the OpenAI API with a defined response structure.
#X–#Z Parse and display the AI's output, including confidence and reasoning.

This AI-based approach offers a much simpler path than the traditional Python version. There's no need to write a normalization function or set up fuzzy logic, just define a BaseModel, craft a clear prompt, pass in your data, and let the model do the reasoning. The entire implementation is conversational, reducing both code overhead and complexity.

The real win here is the AI's reasoning. Along with the best match, it gives you a human-readable explanation of why it chose that record. This isn't something traditional algorithms can offer, and it can surface useful insights or hidden patterns you weren't actively looking for.

But that flexibility cuts both ways. In this case, the AI matched Johnny S. instead of John Smith, interpreting nickname variation and domain similarity.

```
Best match: Johnny S. (johnny@acmeco.com)

Confidence: 0.95

Reasoning:

The incoming email 'jonny_smith@acmeco.com' closely resembles
'johnny@acmeco.com' in both local part and domain. The local part
'jonny_smith' seems like a variation of 'johnny', which is a common nickname
```

`for 'John' or 'Jonathan', and shares the same last name structure. The domain '@acmeco.com' is identical, furthering the match strength. The slight spelling variation aside, this indicates a very high probability that both emails belong to the same customer, Johnny S.`

That's a valid interpretation but it highlights a risk: if you don't set strict rules, the model may improvise. LLMs excel at reasoning, but without clear guardrails, they might go beyond your intent.

Try it now

Replace the incoming email in the AI example with a variation that matches Jane Smythe—for example:

`j.smythe@alpha.io`, `jane_smyth@alphai.io`, or `jsmythe@alpha.io`.

Update the system prompt and input payload accordingly, then run the AI resolution again.

- What match did the AI choose?
- How confident was it in its selection?
- Did the reasoning make sense?
- Does this increase or decrease your trust in the AI's matching ability?

This quick test gives you a feel for how AI handles ambiguity and whether its flexible reasoning feels like an asset or a liability.

7.4 Time Series and Date-Time Transformations

Date and time handling is one of the most deceptively complex challenges in data engineering. While basic timezone conversions were covered in Chapter 6, real-world scenarios often demand more: transforming a single timestamp into multiple formats for reporting, parsing fields like quarter or fiscal period, and running time-based calculations that align with business calendars. These tasks aren't just technical chores, they power everything from payment forecasting to executive dashboards.

In this section, we'll take on a practical challenge from the world of accounts receivable. You've got a list of transaction records with balances due, each tagged with a transaction date and payment terms like NET30 or NET60. Your job: standardize those dates across time zones, break them down into meaningful components (like fiscal quarter), and calculate when each payment is due. We'll also roll up transaction data across time windows to support aging analysis and other business-critical reporting

```
transactions = [
    {"account": "A001", "transaction_date": "2025-01-31T16:00:00Z", "terms": "NET30", "amount_due": 1200},
    {"account": "A001", "transaction_date": "2025-02-28T12:45:00Z", "terms": "NET60", "amount_due": 800},
    {"account": "A001", "transaction_date": "2025-03-15T09:30:00Z", "terms": "NET30", "amount_due": 1500},
    {"account": "A001", "transaction_date": "2025-06-01T11:00:00Z", "terms": "NET15", "amount_due": 950},
    {"account": "A001", "transaction_date": "2025-07-04T10:00:00Z", "terms": "NET30", "amount_due": 700},

    {"account": "A002", "transaction_date": "2025-01-10T14:00:00Z", "terms": "NET45", "amount_due": 300},
    {"account": "A002", "transaction_date": "2025-02-12T08:30:00Z", "terms": "NET30", "amount_due": 600},
    {"account": "A002", "transaction_date": "2025-03-29T17:15:00Z", "terms": "NET60", "amount_due": 1100},
    {"account": "A002", "transaction_date": "2025-04-20T13:00:00Z", "terms": "NET30", "amount_due": 950},
    {"account": "A002", "transaction_date": "2025-06-15T07:00:00Z", "terms": "NET30", "amount_due": 800},

    {"account": "A003", "transaction_date": "2025-03-01T18:30:00Z", "terms": "NET30", "amount_due": 400},
    {"account": "A003", "transaction_date": "2025-04-01T09:00:00Z", "terms": "NET90", "amount_due": 2500},
    {"account": "A003", "transaction_date": "2025-05-15T11:30:00Z", "terms": "NET30", "amount_due": 600},
    {"account": "A003", "transaction_date": "2025-07-01T08:00:00Z", "terms": "NET60", "amount_due": 1000},
    {"account": "A003", "transaction_date": "2025-08-05T16:45:00Z", "terms": "NET30", "amount_due": 750},

    {"account": "A004", "transaction_date": "2025-02-20T15:00:00Z", "terms": "NET15", "amount_due": 900},
    {"account": "A004", "transaction_date": "2025-03-15T10:45:00Z", "terms": "NET30", "amount_due": 850},
    {"account": "A004", "transaction_date": "2025-04-10T14:15:00Z", "terms": "NET45", "amount_due": 1200},
    {"account": "A004", "transaction_date": "2025-06-30T09:00:00Z", "terms": "NET30", "amount_due": 1000},
    {"account": "A004", "transaction_date": "2025-09-01T13:20:00Z", "terms": "NET60", "amount_due": 1300}
]
```

Figure 7.3: Sample list of transactions we will use for time series and advanced date transformation examples using Python and AI.

In this exercise, we'll enhance each transaction_date by generating multiple representations: a standard date format (no timestamp), timestamps in PST, EST, and GST, as well as the extracted month and year. We'll also derive a custom fiscal quarter based on the company's internal calendar, where Q1 spans February to April. Next, we'll calculate the due date for each transaction by applying the payment terms (NET30, NET60, etc.) using business days only, not calendar days. Finally, we'll compute each transaction's percentage contribution to the total outstanding balance for its account, critical for AR aging analysis and prioritization.

Listing 7.7: Python - Time Series and Date-Time Transformations

```
import pandas as pd  #A
from pandas.tseries.offsets import BDay  #B
import pytz  #C
from datetime import datetime  #D

# Load transaction data into a DataFrame  #E
df = pd.DataFrame(transactions)  #F
df["transaction_date"] = pd.to_datetime(df["transaction_date"], utc=True)  #G

# A: Plain date without time  #H
df["date_only"] = df["transaction_date"].dt.date  #I

# B–D: Timestamps converted to PST, EST, and GST respectively  #J
df["timestamp_pst"] = df["transaction_date"].dt.tz_convert("US/Pacific")  #K
df["timestamp_est"] = df["transaction_date"].dt.tz_convert("US/Eastern")  #L
df["timestamp_gst"] = df["transaction_date"].dt.tz_convert("Asia/Dubai")  #M

# E: Extracted month and year for reporting breakdowns  #N
df["month"] = df["transaction_date"].dt.month  #O
df["year"] = df["transaction_date"].dt.year  #P

# F: Custom fiscal quarter based on internal calendar (Q1 = Feb–Apr)  #Q
def get_fiscal_quarter(date):  #R
    fiscal_month = (date.month - 1) % 12 + 1  #S
    if fiscal_month in [2, 3, 4]: return "Q1"  #T
```

```
    elif fiscal_month in [5, 6, 7]: return "Q2"  #U
    elif fiscal_month in [8, 9, 10]: return "Q3"  #V
    else: return "Q4"  #W

df["fiscal_quarter"] = df["transaction_date"].apply(get_fiscal_quarter)  #X

# 2: Due date calculation using business days only  #Y
def get_due_date(row):  #Z
    term_days = int(row["terms"].replace("NET", ""))  #AA
    return row["transaction_date"] + BDay(term_days)  #AB

df["due_date"] = df.apply(get_due_date, axis=1)  #AC

# 3: Percent contribution of each transaction to its account's total balance  #AD
account_totals = df.groupby("account")["amount_due"].transform("sum")  #AE
df["contribution_pct"] = (df["amount_due"] / account_totals * 100).round(2)  #AF

# Display key columns for verification  #AG
display(df)
```

#A–#G: Import libraries, load data, and prepare datetime objects.
#H–#P: Extract a clean date, timezones (PST/EST/GST), and reporting breakdowns.
#Q–#X: Apply a custom fiscal quarter based on a 1-month offset.
#Y–#AC: Calculate business-day due dates based on NET terms.
#AD–#AF: Determine each transaction's share of the account's total balance.
#AG–#AH: Output essential transformation results for verification.

While this implementation successfully handles the necessary transformations, it highlights the overhead involved in traditional techniques. We had to import specific libraries like BDay just to handle business-day arithmetic—a nuance that could easily be missed without deep pandas familiarity. Bespoke functions were required to calculate both the company's custom fiscal quarter and the business-aware due dates. Additionally, we created a separate account_totals object just to compute each transaction's share of its account's balance. While effective, this approach demands precise technical knowledge, manual setup, and domain-specific logic. In the next section, we'll see how an AI-powered, conversational approach simplifies these same operations with less code and far greater flexibility.

Listing 7.8: AI - Time Series and Date-Time Transformations

```
import openai  #A
import os  #B
from dotenv import load_dotenv  #C
from pydantic import BaseModel  #D
from datetime import datetime  #E
import pandas as pd  #F

load_dotenv()  #G
openai.api_key = os.getenv("OPENAI_API_KEY")  #H

# Define response schema using Pydantic  #I
class TransformedTransaction(BaseModel):  #J
    account: str  #K
    transaction_date: str  #L
    date_only: str  #M
    timestamp_pst: str  #N
    timestamp_est: str  #O
    timestamp_gst: str  #P
    month: int  #Q
    year: int  #R
    fiscal_quarter: str  #S
    due_date: str  #T
    contribution_pct: float  #U
```

```
# Compute account totals for contribution_pct  #W
df = pd.DataFrame(transactions)  #X
account_totals = df.groupby("account")["amount_due"].sum().to_dict()  #Y

# Define system prompt template  #Z
system_prompt = f"""
You are a data transformation assistant. You will receive one transaction at a time and
must return a single object matching this schema:
{TransformedTransaction.schema_json(indent=2)}

The transaction will contain:
- 'transaction_date': an ISO-8601 timestamp in UTC
- 'terms': in format like 'NET30', meaning due in 30 **business days**
- 'account': account name
- 'amount_due': numeric value in USD
- 'account_total': total balance due for the account (used for calculating percentage
contribution)

For each transaction:
- Extract the date without time
- Convert the timestamp to PST, EST, and GST
- Extract the month and year
- Determine fiscal quarter using a custom calendar: Q1 = Feb-Apr, Q2 = May-Jul, Q3 =
Aug-Oct, Q4 = Nov-Jan
- Calculate due date by adding the NET days as business days to the transaction date
- Calculate contribution percentage = (amount_due / account_total) * 100

Return a JSON object that matches the schema exactly.
""".strip()  #AA

# Collect results  #AB
results = []  #AC

for tx in transactions:  #AD
    payload = tx.copy()  #AE
    payload["account_total"] = account_totals[tx["account"]]  #AF

    try:  #AG
        completion = openai.beta.chat.completions.parse(  #AH
            model="gpt-4o",  #AI
            messages=[  #AJ
                {"role": "system", "content": system_prompt},  #AK
                {"role": "user", "content": f"{payload}"}  #AL
            ],
            response_format=TransformedTransaction  #AM
        )
        results.append(completion.choices[0].message.parsed.dict())  #AN
    except Exception as e:  #AO
        print(f"Error: {e}")  #AP

# Final result as DataFrame  #AQ
final_df = pd.DataFrame(results)  #AR
display(final_df)  #AS
```

#A–#H: Import libs, load env, and set up OpenAI.
#I–#U: Define the response schema with Pydantic.
#W–#Y: Load data and compute account totals.
#Z–#AA: Craft system prompt with transformation rules.
#AB–#AC: Init result list.
#AD–#AF: Loop over transactions and attach totals.
#AG–#AP: Call API, parse response, handle errors.
#AQ–#AS: Convert results to DataFrame and display.

While the AI approach required a slightly more detailed system prompt and some upfront aggregation logic in Python (like computing per-account totals), the overall pattern stayed true to our standard template. We defined a clear Pydantic response schema, looped through

each transaction, and fed in the necessary context—"role": "user", "content": f"{payload}"—to guide the model's reasoning.

Despite these small deviations, this method remains vastly simpler, more flexible, and far more conversational than the traditional Python implementation. And best of all, both approaches yield identical, verifiable results.

Try it now

Let's give you a little hands-on action, sha. Update the AI script to add support for one or more of the following:

- A new timezone (e.g., CET or JST).
- A different custom fiscal calendar (e.g., Q1 = Mar–May, Q2 = Jun–Aug, etc.).
- A new derived column like "day of the week" or a business-friendly "reporting period."

To do this, you'll need to:

- Modify the TransformedTransaction Pydantic class to include your new fields.
- Update the system_prompt string to clearly instruct the AI how to calculate these new values.

Compare this experience to what you'd do with pure Python:

- Would you need to research time zone offsets?
- Write custom logic for a new fiscal calendar mapping?
- Manually manage data integrity across new columns?

Ask yourself: was the AI implementation easier to modify? More intuitive to reason about? Or did the flexibility come with complexity?

7.5 Lab

This lab builds on the advanced data transformation techniques you practiced throughout Chapter 7—handling nested data structures, complex text processing with regex, string normalization and entity resolution, and time series transformations with AI.

You'll work with three datasets representing TheraGPT's messy CRM system: incoming_customers.json (nested customer onboarding data), crm_customers.csv (existing CRM records with duplicates), and transactions.csv (financial data with timezone issues).

Your goal is to clean, merge, and analyze this data using both traditional Python and AI-driven solutions. If you want to see the full workflow step by step, open the Chapter 7 Lab Jupyter Notebook—it contains working code, prompt examples, and helpful outputs to follow along.

The datasets include several real-world data engineering challenges:

Nested Data Structures: Deeply nested JSON from mobile onboarding with personal info, addresses, subscription details, and therapy preferences buried in multiple levels.

Complex Text Processing: Messy address lines (123mainstreet,newyorkcity,ny,10000) and

subscription codes (GOLD2xONLINE) that need regex parsing into structured components.

Entity Resolution: CRM records where the same person appears multiple times with slight variations (Alexander vs Alex Johnson, different email formats, address inconsistencies).

ime Series Transformations: Transaction timestamps in various formats and timezones, NET payment terms that need business-day calculations, and fiscal quarter mappings.

Golden Record Creation: Merging duplicate customer records and building comprehensive account profiles for B2B analysis.

1. Flatten Nested JSON Using Python and AI

Extract customer data from deeply nested JSON structures using both json_normalize and AI-powered structured extraction.

2. Parse Complex Text Fields Using Regex and AI

Transform messy text fields into structured data:

- Address parsing: Extract street, city, state, zip from various formats
- Subscription decoding: Parse tier, frequency, and channel from codes like PLATINUM2xHYBRID

3. Entity Resolution and Golden Records

Identify duplicate customers across the CRM database and create consolidated golden records:

- Normalize customer data (names, emails, addresses, states, zip codes)
- Use AI to identify potential duplicates based on email similarity, name variations, and address matching
- Merge duplicate records into golden records with the best information from each source
- Match incoming customers to existing golden records

4. Time Series and Financial Data Transformations

Clean and transform transaction data:

- Convert timestamps across multiple timezones (PST, EST, GST)
- Calculate due dates using business days and NET terms
- Apply custom fiscal calendar mappings
- Calculate contribution percentages by account

5. Extra Credit: Build Golden Account List for B2B Analysis

Create comprehensive account profiles by aggregating customer and transaction data:

- Match emails to company domains
- Calculate total spend, user counts, and business opportunity scores
- Identify high-value B2B prospects

7.6 Lab Answers

Refer to the Chapter 7 Lab Jupyter Notebook for full answers.

1. Flattening Nested JSON with AI

The AI approach uses structured response parsing to extract nested data reliably:

```
import openai
import os
from dotenv import load_dotenv
from pydantic import BaseModel
from typing import Optional, List
from tqdm.notebook import tqdm

# Load API key
load_dotenv()
openai.api_key = os.getenv('OPENAI_API_KEY')

# Define the flattened structure
class FlattenedCustomer(BaseModel):
    customer_id: str
    first_name: str
    last_name: str
    email: str
    address_line: str
    apartment: Optional[str]
    building: Optional[str]
    subscription_code: str
    subscription_start: str
    preferred_time: str
    therapist_gender: str
    therapy_topics: str  # Comma-separated string

# System prompt for flattening
system_prompt = """
You are a data extraction assistant. Extract and flatten the nested customer data into the
following structure:
- customer_id: The customer ID
- first_name: From profile.personal.first_name
- last_name: From profile.personal.last_name
- email: From profile.personal.email
- address_line: From profile.address.address_line
- apartment: From profile.address.details.apartment (null if not present)
- building: From profile.address.details.building (null if not present)
- subscription_code: From profile.subscription.subscription_code
- subscription_start: From profile.subscription.start_date
- preferred_time: From profile.therapy_preferences.preferred_time
- therapist_gender: From profile.therapy_preferences.therapist_gender
- therapy_topics: Join the topics array into a comma-separated string

Return the result as a JSON object matching the FlattenedCustomer structure.
"""
```

```
# Process each record with AI
ai_flattened = []
for record in tqdm(nested_data, desc="AI Flattening"):
    try:
        completion = openai.beta.chat.completions.parse(
            model="gpt-4o",
            messages=[
                {"role": "system", "content": system_prompt},
                {"role": "user", "content": str(record)}
            ],
            response_format=FlattenedCustomer
        )
        ai_flattened.append(completion.choices[0].message.parsed.dict())
    except Exception as e:
        print(f"Error processing record: {e}")
        continue

df_ai = pd.DataFrame(ai_flattened)
print(f"\nFlattened {len(df_ai)} customer records using AI:")
display(df_ai)
```

This approach handles schema changes gracefully—if new nested fields are added, you simply update the prompt and data class without rewriting complex extraction logic.

2. Complex Text Processing with AI

AI excels at parsing messy text formats that would require complex regex patterns:

```
# Define structure for parsed fields
class ParsedFields(BaseModel):
    street: Optional[str]
    city: Optional[str]
    state: Optional[str]
    zip_code: Optional[str]
    subscription_tier: Optional[str]
    subscription_frequency: Optional[str]
    subscription_channel: Optional[str]

# System prompt for parsing
parse_prompt = """
You are a text parsing assistant. Parse the given address_line and subscription_code
fields:

For address_line:
- Extract street, city, state, and zip_code
- Handle formats like "123mainstreet,newyorkcity,ny,10000" or "456 Elm St Boston
Massachusetts 02118"

For subscription_code:
- Extract tier, frequency, and channel from codes like "GOLD2xONLINE"
- Format frequency as "2x per week" style
- Tier should be the first part (GOLD, SILVER, PLATINUM)
- Channel should be the last part (ONLINE, INPERSON)

Return the result as a JSON object matching the ParsedFields structure.
"""

# Process each record with AI
ai_parsed = []
for _, row in tqdm(df_ai.iterrows(), total=len(df_ai), desc="AI Parsing"):
    input_data = {
        "address_line": row['address_line'],
        "subscription_code": row['subscription_code']
    }
```

```
    try:
        completion = openai.beta.chat.completions.parse(
            model="gpt-4o",
            messages=[
                {"role": "system", "content": parse_prompt},
                {"role": "user", "content": str(input_data)}
            ],
            response_format=ParsedFields
        )
        parsed = completion.choices[0].message.parsed.dict()
        ai_parsed.append(parsed)
    except Exception as e:
        print(f"Error parsing record: {e}")
        ai_parsed.append({
            'street': None, 'city': None, 'state': None, 'zip_code': None,
            'subscription_tier': None, 'subscription_frequency': None,
'subscription_channel': None
        })

# Add parsed fields to AI dataframe
df_ai_parsed = df_ai.copy()
for i, parsed in enumerate(ai_parsed):
    for key, value in parsed.items():
        df_ai_parsed.loc[i, key] = value

print("\nParsed data using AI:")
display(df_ai_parsed[['customer_id', 'first_name', 'last_name', 'street', 'city', 'state',
'zip_code',
                      'subscription_tier', 'subscription_frequency',
'subscription_channel']])
```

The AI approach adapts to format variations automatically and can handle new subscription types like ENTERPRISE3xHYBRID without code changes.

3. AI-Powered Entity Resolution

Instead of rule-based matching, AI performs sophisticated entity resolution by considering multiple factors:

```
class EntityMatch(BaseModel):
    customer_id: str
    similar_customers: List[str]
    match_confidence: float
    match_reasoning: str
    is_likely_duplicate: bool
entity_resolution_prompt = """
You are an entity resolution specialist. Identify if this customer appears to be the same
person as any other customers:
Consider these matching factors:
1. Email similarity (exact match is strongest indicator)
2. Name similarity (nicknames, abbreviations, typos)
3. Address similarity (same location, minor formatting differences)
Rules for matching:
- Exact email matches are very strong indicators
- Similar names + similar addresses are strong indicators
- Consider common nicknames (Alexander/Alex, Jordan/J)
- Consider typos and formatting differences
- Be conservative - only flag as duplicate if confidence is high (>0.7)
"""
```

The AI evaluates each customer against all others, providing confidence scores and detailed reasoning for matches. This handles edge cases like nickname variations and email format differences that rule-based systems often miss.

4. Golden Record Creation

AI consolidates duplicate records by intelligently choosing the best information from each source:

```
class GoldenRecord(BaseModel):
    golden_id: str
    customer_ids: List[str]
    first_name: str
    last_name: str
    email: str
    email_domain: str
    address: str
    state: str
    zip_code: str
    subscription_code: str
    confidence_score: float
    source_records: int
    is_business_email: bool
golden_prompt = """
You are a data consolidation assistant. Merge these duplicate customer records into a
single "golden record":
Rules for consolidation:
- Use the most complete and accurate information available
- For names: prefer full names over nicknames, but use consistent capitalization
- For emails: prefer the most complete/valid email address
- For addresses: use the most complete address information
- For states: normalize to 2-letter abbreviations (NY, CA, FL, etc.)
- For zip codes: use 5-digit format
- Extract email domain from the email address
- Determine if this is a business email (corporate domain vs gmail/yahoo/hotmail)
- Assign a confidence_score (0.0-1.0) based on data completeness and consistency
"""
```

This creates high-quality master records that combine the best attributes from multiple source records.

5. Time Series Transformations with AI

AI handles complex date-time transformations and business logic:

```
class TransformedTransaction(BaseModel):
    transaction_id: str
    account: str
    user_email: str
    date_only: str
    timestamp_pst: str
    timestamp_est: str
    timestamp_gst: str
    month: int
    year: int
    fiscal_quarter: str
    due_date: str
    contribution_pct: float
time_prompt = """
You are a data transformation assistant specializing in time series and financial data:
Required transformations:
- date_only: Extract just the date (YYYY-MM-DD format)
- timestamp_pst: Convert to Pacific Standard Time
- timestamp_est: Convert to Eastern Standard Time
- timestamp_gst: Convert to Gulf Standard Time (Asia/Dubai)
```

```
- fiscal_quarter: Calculate using custom fiscal calendar where Q1=Feb-Apr, Q2=May-Jul,
Q3=Aug-Oct, Q4=Nov-Jan
- due_date: Add the NET term days as BUSINESS DAYS (not calendar days) to the transaction
date
- contribution_pct: Calculate (amount_due / account_total) * 100, rounded to 2 decimal
places
"""
```

The AI correctly handles timezone conversions, business day calculations, and custom fiscal calendars without requiring complex date manipulation libraries.

6. Golden Account Analysis

AI creates comprehensive B2B account profiles by analyzing combined customer and transaction data:

```
class GoldenAccount(BaseModel):
    account_name: str
    company_domain: str
    total_users: int
    total_spend: float
    avg_spend_per_user: float
    primary_location: str
    subscription_tiers: List[str]
    business_opportunity_score: float
    account_status: str
    key_contacts: List[str]
account_prompt = """
You are a B2B account analysis assistant. Create a comprehensive golden account record:
Your task:
1. Analyze all users associated with this account
2. Calculate total spend, user count, and average spend per user
3. Identify the primary business location (most common state/city)
4. List unique subscription tiers used by this account
5. Calculate a business opportunity score (0-100) based on:
   - Total spend (higher = better)
   - Number of users (more users = better scaling potential)
   - Subscription diversity (multiple tiers = growth potential)
   - Business email usage (corporate domains = B2B opportunity)
6. Assign account status: "High Value" (score 80+), "Growth Potential" (score 50-79),
"Standard" (score <50)
"""
```

This produces actionable B2B intelligence by synthesizing customer demographics, spending patterns, and engagement metrics into prioritized account lists.The key advantage of the AI approach throughout this lab is adaptability—as data formats change, new subscription types are added, or business rules evolve, you update prompts rather than rewriting complex parsing and transformation logic. This makes AI-driven data engineering pipelines more maintainable and scalable for production environments.