Python Refresher for LLM Application Development

This Python refresher covers the absolute fundamentals you must know to succeed in this course—while you may have forgotten some of these concepts, you're expected to understand them completely and recognize what they do, as these are the essential building blocks for everything we'll cover. Additionally, we introduce modern features like type hints that you may not be familiar with, which represent current Python best practices we'll use throughout the class.

Why Python for AI Applications?

Python has become the go-to language for AI and machine learning projects because:

- **Simple syntax**: Reads almost like English
- **Rich ecosystem**: Thousands of libraries for AI, web development, and data processing
- **Strong community**: Extensive documentation and support
- **Industry standard**: Most LLM APIs and frameworks are designed with Python in mind

What You'll Learn

This refresher covers the essential Python concepts:

- **Basic Data Types**: Numbers, text, and boolean values—the building blocks of all programs
- **Data Structures**: Lists, dictionaries, and other containers for organizing information
- **Program Flow**: Making decisions and repeating actions in your code
- **Functions**: Creating reusable pieces of code (crucial for clean LLM applications)
- **Type Hints**: A modern Python feature that makes your code more reliable and AI-assistant-friendly

1. Basic Data Types

Think of data types as different categories of information your program can work with. Just like in real life, we handle numbers differently than we handle text—Python does the same thing.

Numbers: Integers and Floats

Integers are whole numbers (no decimal points), while **floats** are decimal numbers.

In [5]:
```python
# Integer examples
user_age = 25
items_in_cart = 3
year = 2024

# Float examples
price = 19.99
temperature = 98.6
discount_rate = 0.15  # 15% as a decimal
```

Text: Strings

Strings represent text data and are enclosed in quotes. You can use single quotes (') or double quotes (").

In [ ]:
```python
# String examples
user_message = "Hello, can you help me with Python?"
ai_model_name = 'gpt-4'
system_prompt = """You are a helpful assistant that specializes in
explaining programming concepts to beginners."""

# Strings can contain any characters
email = "user@example.com"
file_path = "C:\\Documents\\my_project.py"
```

> **Tip:** Use triple quotes (`"""` or `'''`) for multi-line strings, which are perfect for longer strings, such as prompts to LLMs.

Boolean Values: True and False

Booleans represent yes/no, on/off, or true/false values. They're essential for making decisions in your code.

In [ ]:
```python
# Boolean examples
is_authenticated = True
has_premium_account = False
api_request_successful = True
```

The Special Value: None

None represents "nothing" or "no value." It's Python's way of saying "this variable exists but doesn't contain meaningful data right now."

In [ ]:
```
user_response = None   # User hasn't responded yet
error_message = None   # No error has occurred
cached_result = None   # No cached data available
```

If you are wondering when you'll use `None` in LLM apps, it's often used for optional parameters, error handling, or when waiting for user input or API responses.

In [8]:
```
age = 30
name = "Alice"
is_student = False

print(type(age))        # <class 'int'>
print(type(name))       # <class 'str'>
print(type(is_student)) # <class 'bool'>
```

<class 'int'>
<class 'str'>
<class 'bool'>

> **Tip:** When building LLM applications, use `type()` to verify you're sending the right kind of data to a function. Many errors occur when you accidentally send a number as text or vice versa.

Converting Between Types

Sometimes you need to convert data from one type to another. Python makes this straightforward:

In [13]:
```
# Converting strings to numbers
user_input = "25"          # This is text
age = int(user_input)      # Convert to integer: 25
print(type(user_input))
print(type(age))
```

<class 'str'>
<class 'int'>

In [ ]:
```
price_text = "19.99"
price = float(price_text)   # Convert to float: 19.99

print(type(price_text))
print(type(price))
```

<class 'str'>
<class 'float'>

In [17]:
```
# Converting numbers to strings
token_count = 150
message = "You used " + str(token_count) + " tokens"
print(message)
```

You used 150 tokens

Understanding "Truthy" and "Falsy" Values

This concept is fundamental to Python and will appear frequently in your LLM applications. Every value in Python can be evaluated as either True or False in a boolean context, even if it's not explicitly a boolean. This implicit conversion happens automatically in conditional statements and is incredibly useful for checking if data exists or is valid.

Values that are "falsy" (evaluate to False):

- `False` (the boolean)
- `0` (zero)
- `""` (empty string)
- (empty collecitons, such as lists of dictionary)
- `None`

Everything else is "truthy" (evaluates to True):

- Non-empty strings, lists, dictionaries
- Any non-zero number (including negative numbers)
- Most objects and data structures

In [18]:
```
# Converting to boolean
empty_string = ""
print(bool(empty_string))   # False (empty strings are "falsy")
non_empty = "Hello"
print(bool(non_empty))      # True (non-empty strings are "truthy")
```

False
True
In [20]:`bool("")`

Out[20]:False
In [21]:`bool("Hello")`

Out[21]:True
In [22]:`bool("0")`

Out[22]:True
In [23]:`bool(0)`

Out[23]:False

String Formatting: Making Text Dynamic

When building LLM applications, you'll constantly need to create dynamic text—combining variables with fixed text to create prompts, messages, or responses.

Python offers several ways to format strings, but we'll focus on the most modern and readable approach: **f-strings** (formatted string literals).

F-String Formatting (Recommended)

F-strings, introduced in Python 3.6, are the most readable and efficient way to format strings. Simply put an `f` before your quotes and use curly braces `{}` to include variables:

In [ ]:
```python
# Basic f-string usage
user_name = "Sarah"
message_count = 42
model_name = "gpt-4"

# Create dynamic messages
greeting = f"Welcome back, {user_name}!"
status = f"You have {message_count} messages in your chat history."
model_info = f"Currently using {model_name} for responses."

print(greeting)    # "Welcome back, Sarah!"
print(status)      # "You have 42 messages in your chat history."
print(model_info)  # "Currently using gpt-4 for responses."
```

Welcome back, Sarah!
You have 42 messages in your chat history.
Currently using gpt-4 for responses.

F-Strings with Expressions

You can put any Python expression inside the curly braces:

In [ ]:
```python
# Math operations
tokens_used = 150
max_tokens = 2000
remaining = f"You have {max_tokens - tokens_used} tokens remaining."
print(remaining)  # "You have 1850 tokens remaining."

# Method calls
user_email = "SARAH@EXAMPLE.COM"
formatted_email = f"User email: {user_email.lower()}"
print(formatted_email)  # "User email: sarah@example.com"

# Conditional expressions
temperature = 0.7
creativity_level = f"Creativity: {'High' if temperature > 0.5 else 'Low'}"
print(creativity_level)  # "Creativity: High"
```

You have 1850 tokens remaining.
User email: sarah@example.com
Creativity: High

Formatting Numbers in F-Strings

F-strings provide powerful options for formatting numbers:

In [ ]:
```python
# Decimal places
price = 19.99999
formatted_price = f"Cost: ${price:.2f}"  # 2 decimal places
print(formatted_price)
```

Cost: $20.00
The `:.2f` format specifier in the f-string tells Python to display the number as a float with exactly 2 decimal places. When you specify fewer decimal places than the number contains, Python automatically rounds to the nearest value. In this case, 19.99999 gets rounded to 20.00 because the third decimal place (9) causes rounding up. The `f` stands for `float` and the 2 specifies exactly 2 digits after the decimal point.

Percentages

In [ ]:
```python
accuracy = 0.8547
formatted_accuracy = f"Model accuracy: {accuracy:.1%}"  # 1 decimal percentage
print(formatted_accuracy)  # "Model accuracy: 85.5%"
```

Model accuracy: 85.5%

Large numbers with commas

In [ ]:
```python
total_tokens = 1234567
formatted_tokens = f"Total processed: {total_tokens:,} tokens"
print(formatted_tokens)  # "Total processed: 1,234,567 tokens"
```

Total processed: 1,234,567 tokens

Multi-line F-Strings for LLM Prompts

F-strings work great for creating complex prompts for LLMs:

In [32]:
```python
user_question = "What's the weather like?"
user_location = "New York"
current_time = "2:30 PM"

# Multi-line f-string for a detailed prompt
system_prompt = f"""You are a helpful weather assistant.

Current context:
- User location: {user_location}
- Current time: {current_time}
- User question: "{user_question}"

Please provide a helpful and accurate response about the weather.
Include current conditions and a brief forecast if possible."""

print(system_prompt)
```

You are a helpful weather assistant.

Current context:
- User location: New York
- Current time: 2:30 PM
- User question: "What's the weather like?"

Please provide a helpful and accurate response about the weather.
Include current conditions and a brief forecast if possible.

Other Formatting Methods (For Reference)

While f-strings are recommended, you might encounter these other methods in existing code:

In [34]:
```python
name = "Alice"
age = 30

# .format() method (older but still valid)
message1 = "Hello, {}! You are {} years old.".format(name, age)
message2 = "Hello, {name}! You are {age} years old.".format(name=name, age=age)

# % formatting (oldest, rarely used now)
message3 = "Hello, %s! You are %d years old." % (name, age)
print("First method: "+ message1)
print("First method: "+ message2)
print("First method: "+ message3)
```

First method: Hello, Alice! You are 30 years old.
First method: Hello, Alice! You are 30 years old.
First method: Hello, Alice! You are 30 years old.

while all three approaches can be used interchangeably, F-strings are considered the most Pythonic string formatting method for several important reasons:

1. Readability: Variables appear directly where they'll be used in the string

```
# F-string - clear and intuitive
message = f"Hello, {name}! You are {age} years old."

# .format() - variables separated from their position
message = "Hello, {}! You are {} years old.".format(name, age)
```

2. Performance: F-strings are faster because they're evaluated at runtime rather than requiring method calls

3. Less Error-Prone: No need to match variable order with placeholder positions

4. Conciseness: Shorter and more direct syntax

5. Expression Support: You can include calculations directly in the braces

```
f"You'll be {age + 1} next year"  # Clean and readable
```

The older methods (`.format()` and `%` formatting) still work and you'll see them in legacy code, but f-strings are the modern standard. For LLM applications where you're frequently building dynamic prompts and messages, f-strings make your code much more maintainable and readable.

> **Tip:** Stick with f-strings for new code. They're more readable, faster, and less error-prone.

2. Data Structures: Organizing Your Information

When building applications, you'll need to organize and manipulate data efficiently. Python provides several built-in data structures that are perfect for common tasks like storing conversation history, managing user preferences, or handling API responses.

Lists: Ordered Collections of Items

Lists are ordered collections that can hold any type of data. Think of them as digital filing cabinets where you can store items in a specific order and easily add, remove, or modify them.

Creating Lists

List can be created using various methods

In [46]:
```
# Empty list
chat_history = []

chat_history
```

Out[46]:[]

In [45]:
```
# List with initial values
ai_models = ["gpt-3.5-turbo", "gpt-4", "claude-3", "gemini-pro"]
ai_models
```

Out[45]:['gpt-3.5-turbo', 'gpt-4', 'claude-3', 'gemini-pro']

In [43]:
```
# Mixed data types (though this is less common)
user_data = ["Alice", 25, True, "premium"]
user_data
```

Out[43]:['Alice', 25, True, 'premium']

In [42]:
```
# List from other data (very useful!)

user_input = "Hello, how are you today?"
words = list(user_input.split())
words
```

Out[42]:['Hello,', 'how', 'are', 'you', 'today?']

In [37]:
```
# List with repeated values
default_scores = [0.0] * 5
default_scores
```

Out[37]:[0.0, 0.0, 0.0, 0.0, 0.0]

Accessing List Elements

Lists use **zero-based indexing**, meaning the first item is at position 0:

In [48]:
```
models = ["gpt-3.5-turbo", "gpt-4", "claude-3", "gemini-pro"]

# Positive indexing (from the start)
first_model = models[0]      # "gpt-3.5-turbo"
second_model = models[1]     # "gpt-4"
```

```python
# Negative indexing (from the end)
last_model = models[-1]      # "gemini-pro"
second_last = models[-2]     # "claude-3"

print(f"First model: {first_model}")
print(f"Last model: {last_model}")
```

First model: gpt-3.5-turbo
Last model: gemini-pro

List Slicing: Getting Portions of Lists

Slicing lets you extract portions of a list using the syntax `list[start:stop:step]` :

In [2]:
```python
conversation = ["Hello", "Hi there!", "How can I help?", "I need coding help", "Sure, what language?",

# Basic slicing
first_three = conversation[0:3]    # ["Hello", "Hi there!", "How can I help?"]
first_three
```

Out[2]:['Hello', 'Hi there!', 'How can I help?']

In [55]:
```python
last_two = conversation[-2:]        # ["Python please"]
last_two
```

Out[55]:['Sure, what language?', 'Python please']

In [56]:
```python
middle_part = conversation[2:5]    # ["How can I help?", "I need coding help", "Sure, what language?
middle_part
```

Out[56]:['How can I help?', 'I need coding help', 'Sure, what language?']

In [57]:
```python
# Skip elements
every_other = conversation[::2]    # ["Hello", "How can I help?", "Sure, what language?"]
every_other
```

Out[57]:['Hello', 'How can I help?', 'Sure, what language?']

In [58]:
```python
# Reverse the list
reversed_conv = conversation[::-1] # ["Python please", "Sure, what language?", ...]
reversed_conv
```

Out[58]:['Python please',
'Sure, what language?',
'I need coding help',
'How can I help?',
'Hi there!',
'Hello']

Adding and Removing Items

Lists are **mutable**, meaning you can change them after creation:

In [63]:
```python
# Start with an empty conversation history
messages = []

# Add items to the end
messages.append("User: Hello")
messages.append("AI: Hi! How can I help you today?")
messages.append("User: I need help with Python")

print("After adding:", messages)
```

After adding: ['User: Hello', 'AI: Hi! How can I help you today?', 'User: I need help with Python']

In [64]:
```python
# Add multiple items at once
new_messages = ["AI: I'd be happy to help!", "User: Thanks!"]
messages.extend(new_messages)

print("After extending:", messages)
```

After extending: ['User: Hello', 'AI: Hi! How can I help you today?', 'User: I need help with Python', "AI: I'd be happy to help!", 'User: Thanks!
']

In [65]:
```python
# Insert at a specific position
messages.insert(0, "System: Conversation started")
print("After insert:", messages)
```

After insert: ['System: Conversation started', 'User: Hello', 'AI: Hi! How can I help you today?', 'User: I need help with Python', "AI: I'd be ha
ppy to help!", 'User: Thanks!']

In [66]:
```python
# Remove items
messages.remove("User: Thanks!")  # Remove first occurrence of this value
print("After remove:", messages)
```

After remove: ['System: Conversation started', 'User: Hello', 'AI: Hi! How can I help you today?', 'User: I need help with Python', "AI: I'd be h
appy to help!"]

In [67]:
```python
# Remove by index
```

```python
last_message = messages.pop()     # Remove and return last item
print("Removed:", last_message)
print("Final list:", messages)
```

Removed: AI: I'd be happy to help!
Final list: ['System: Conversation started', 'User: Hello', 'AI: Hi! How can I help you today?', 'User: I need help with Python']

In [68]:
```python
# Remove by index without returning
del messages[0]  # Remove first item
print("After del:", messages)
```

After del: ['User: Hello', 'AI: Hi! How can I help you today?', 'User: I need help with Python']

Useful List Methods

In [69]:
```python
feedback_scores = [4.5, 3.8, 4.9, 4.2, 3.9, 4.5, 4.1]

# Find information about the list
print(f"Number of ratings: {len(feedback_scores)}")
```

Number of ratings: 7

In [70]:
```python
print(f"Highest score: {max(feedback_scores)}")
```

Highest score: 4.9

In [71]:
```python
print(f"Lowest score: {min(feedback_scores)}")
```

Lowest score: 3.8

In [72]:
```python
print(f"Average score: {sum(feedback_scores) / len(feedback_scores):.2f}")
```

Average score: 4.27

In [73]:
```python
# Count occurrences
count_4_5 = feedback_scores.count(4.5)
print(f"Number of 4.5 ratings: {count_4_5}")
```

Number of 4.5 ratings: 2

In [74]:
```python
# Find index of a value
try:
    index_of_highest = feedback_scores.index(4.9)
    print(f"Highest score is at position: {index_of_highest}")
except ValueError:
    print("Value not found in list")
```

Highest score is at position: 2

In [75]:
```python
# Sort the list
feedback_scores.sort()  # Modifies original list
print(f"Sorted scores: {feedback_scores}")
```

Sorted scores: [3.8, 3.9, 4.1, 4.2, 4.5, 4.5, 4.9]

In [76]:
```python
# Create a sorted copy (doesn't modify original)
original_scores = [4.5, 3.8, 4.9, 4.2]
sorted_copy = sorted(original_scores)
print(f"Original: {original_scores}")
print(f"Sorted copy: {sorted_copy}")
```

Original: [4.5, 3.8, 4.9, 4.2]
Sorted copy: [3.8, 4.2, 4.5, 4.9]

List Comprehensions: Powerful One-Liners

List comprehensions provide a concise way to create lists based on existing lists or other iterables. They're incredibly useful in LLM applications for data processing:

In [82]:
```python
# Basic list comprehension
numbers = [1, 2, 3, 4, 5]
squares = [x**2 for x in numbers]  # [1, 4, 9, 16, 25]
squares
```

Out[82]:[1, 4, 9, 16, 25]

```python
squares = [x**2 for x in numbers]
```

You are packing the following logic in this 1 line.

```python
squares = []
for x in numbers:
    squares.append(x**2)
```

In [81]:
```python
# List comprehension with condition
even_squares = [x**2 for x in numbers if x % 2 == 0]  # [4, 16]
even_squares
```

Out[81]:[4, 16]

In [80]:
```python
# Process text data
user_messages = ["Hello there", "HOW ARE YOU?", "thanks for the help"]
lowercase_messages = [msg.lower() for msg in user_messages]
lowercase_messages
```

Out[80]:['hello there', 'how are you?', 'thanks for the help']

In [ ]:
```python
#### TODO: Add this after we discuss the dictionary

# Extract specific data
api_responses = [
    {"model": "gpt-4", "tokens": 150, "cost": 0.03},
    {"model": "gpt-3.5", "tokens": 120, "cost": 0.01},
    {"model": "claude-3", "tokens": 200, "cost": 0.04}
]

# Extract just the costs
costs = [response["cost"] for response in api_responses]  # [0.03, 0.01, 0.04]
costs

# Extract models with high token usage
high_token_models = [resp["model"] for resp in api_responses if resp["tokens"] > 140]
high_token_models
```

Tuples: Immutable Ordered Collections

Tuples are like lists, but they can't be changed after creation. They're perfect for data that shouldn't be modified, like configuration settings or coordinate pairs.

In [ ]:
```python
#### Creating and Using Tuples
```

In [97]:
```python
api_endpoint = ("https://api.openai.com", 443, True)  # (url, port, ssl_enabled)
model_info = ("gpt-4", "GPT-4", 8192)                 # (id, name, max_tokens)
model_info
```

Out[97]:('gpt-4', 'GPT-4', 8192)

In [99]:
```python
# Tuples without parentheses (less clear but correct)
coordinates = 40.7128, -74.0060  # (latitude, longitude)
coordinates
```

Out[99]:(40.7128, -74.006)

In [105]:
```python
# Single item tuple (note the comma!)
single_item = ("gpt-4",)
```

Out[105]:('gpt-4',)

In [106]:
```python
not_a_tuple = ("gpt-4")
not_a_tuple
```

Out[106]:'gpt-4'

In [107]:
```python
# Empty tuple
empty_tuple = ()
empty_tuple
```

Out[107]:()

In [109]:
```python
api_endpoint
```

Out[109]:('https://api.openai.com', 443, True)

In [110]:
```python
# Access elements by unpacking the tuple
url, port, ssl = api_endpoint
print(f"Connecting to {url} on port {port} (SSL: {ssl})")
```

Connecting to https://api.openai.com on port 443 (SSL: True)

In [ ]:
```python
# Indexing works the same as lists
model_id = model_info[0]    # "gpt-4"
max_tokens = model_info[2]  # 8192
```

When to Use Tuples vs Lists

Use tuples for:

- Configuration that shouldn't change
- Return multiple values from functions
- Dictionary keys (must be immutable)
- Representing fixed structures (coordinates, RGB colors, etc.)

Use lists for:

- Data that needs to be modified
- Collections that grow or shrink
- When you need list methods like append(), remove(), etc.

In [111]:
```python
# Good use of tuples
API_CONFIG = ("gpt-4", 0.7, 2000)  # (model, temperature, max_tokens)
RGB_COLORS = {
    "red": (255, 0, 0),
    "green": (0, 255, 0),
```

```
    "blue": (0, 0, 255)
}

# Good use of lists
conversation_messages = []  # Will grow as conversation continues
available_models = ["gpt-4", "gpt-3.5-turbo"]  # Might be updated
```

Dictionaries: Key-Value Data Storage

Dictionaries are Python's way of storing associated data. Think of them as real dictionaries where you look up a word (key) to find its definition (value). They're incredibly useful for LLM applications because most API responses are in dictionary format (JSON).

Creating Dictionaries

In [ ]:
```
# Empty dictionary
user_preferences = {}
user_preferences
```

In [114]:
```
# Dictionary with initial data
ai_model_settings = {
    "model": "gpt-4",
    "temperature": 0.7,
    "max_tokens": 2000,
    "top_p": 1.0
}
ai_model_settings
```

Out[114]:{'model': 'gpt-4', 'temperature': 0.7, 'max_tokens': 2000, 'top_p': 1.0}

In [115]:
```
# Dictionary with mixed value types
user_profile = {
    "name": "Dr. Smith",
    "email": "dr.smith@university.edu",
    "age": 45,
    "is_premium": True,
    "favorite_models": ["gpt-4", "claude-3"],
    "settings": {"theme": "dark", "notifications": True}
}
user_profile
```

Out[115]:
```
{'name': 'Dr. Smith',
 'email': 'dr.smith@university.edu',
 'age': 45,
 'is_premium': True,
 'favorite_models': ['gpt-4', 'claude-3'],
 'settings': {'theme': 'dark', 'notifications': True}}
```

In [116]:
```
# Create dictionary by zipping  lists (very useful!)
keys = ["model", "temperature", "max_tokens"]
values = ["gpt-4", 0.7, 2000]
# we could do
zipped_keys_and_values = [('model', 'gpt-4'), ('temperature', 0.7), ('max_tokens', 2000)]
```

{'model': 'gpt-4', 'temperature': 0.7, 'max_tokens': 2000}

In [118]:
```
config = dict(zip(keys, values))
print(config)  # {"model": "gpt-4", "temperature": 0.7, "max_tokens": 2000}
```

{'model': 'gpt-4', 'temperature': 0.7, 'max_tokens': 2000}

In [120]:
```
# The following automatically "zips" the two lists
config = dict(zip(keys, values))
print(config)  # {"model": "gpt-4", "temperature": 0.7, "max_tokens": 2000}
```

{'model': 'gpt-4', 'temperature': 0.7, 'max_tokens': 2000}

In [ ]:
```
#### Accessing and Modifying Dictionary Values
```

In [123]:
```
# Access values using keys
model_name = ai_model_settings["model"]        # "gpt-4"
temperature = ai_model_settings["temperature"] # 0.7
print(model_name, "-------",temperature)
```

gpt-4 ------- 0.7

In [125]:
```
# Safe access with .get() (returns None if key doesn't exist)
max_length = ai_model_settings.get("max_length")      # None
print(max_length)
```

None

In [126]:
```
max_length = ai_model_settings.get("max_length", 1000) # 1000 (default value)
print(max_length)
```

1000

In [127]:
```
# Modify existing values
ai_model_settings["temperature"] = 0.9
ai_model_settings["model"] = "gpt-3.5-turbo"
```

```
ai_model_settings
```

Out[127]:{'model': 'gpt-3.5-turbo',
 'temperature': 0.9,
 'max_tokens': 2000,
 'top_p': 1.0}

In [131]:
```
# Add new key-value pairs
ai_model_settings["stream"] = True
ai_model_settings["stop_sequences"] = ["\n", "###"]
ai_model_settings
```

Out[131]:{'model': 'gpt-3.5-turbo',
 'temperature': 0.9,
 'max_tokens': 2000,
 'top_p': 1.0,
 'stream': True,
 'stop_sequences': ['\n', '###']}

In [132]:
```
# Remove items
del ai_model_settings["stop_sequences"]  # Remove key-value pair
popped_value = ai_model_settings.pop("stream", False)  # Remove and return value
popped_value
```

Out[132]:True

In [134]:
```
print("After removal:", ai_model_settings)
```

After removal: {'model': 'gpt-3.5-turbo', 'temperature': 0.9, 'max_tokens': 2000, 'top_p': 1.0}

Useful Dictionary Methods

In [136]:
```
api_response = {
    "id": "chatcmpl-123",
    "model": "gpt-4",
    "choices": [{"message": {"content": "Hello! How can I help you today?"}}],
    "usage": {"prompt_tokens": 10, "completion_tokens": 12, "total_tokens": 22}
}

# Get all keys, values, or key-value pairs
print("Keys:", list(api_response.keys()))
print("Values:", list(api_response.values()))
print("Items:", list(api_response.items()))
```

Keys: ['id', 'model', 'choices', 'usage']
Values: ['chatcmpl-123', 'gpt-4', [{'message': {'content': 'Hello! How can I help you today?'}}], {'prompt_tokens': 10, 'completion_tokens': 12
, 'total_tokens': 22}]
Items: [('id', 'chatcmpl-123'), ('model', 'gpt-4'), ('choices', [{'message': {'content': 'Hello! How can I help you today?'}}]), ('usage', {'prompt_to
kens': 10, 'completion_tokens': 12, 'total_tokens': 22})]

In [137]:
```
# Check if key exists
if "usage" in api_response:
    tokens_used = api_response["usage"]["total_tokens"]
    print(f"Tokens used: {tokens_used}")
```

Tokens used: 22

In [139]:
```
# Update with another dictionary
additional_data = {"timestamp": "2024-01-15", "user_id": "user123"}
api_response.update(additional_data)
api_response
```

Out[139]:{'id': 'chatcmpl-123',
 'model': 'gpt-4',
 'choices': [{'message': {'content': 'Hello! How can I help you today?'}}],
 'usage': {'prompt_tokens': 10, 'completion_tokens': 12, 'total_tokens': 22},
 'timestamp': '2024-01-15',
 'user_id': 'user123'}

In [140]:
```
# Create a copy
response_backup = api_response.copy()
response_backup
```

Out[140]:{'id': 'chatcmpl-123',
 'model': 'gpt-4',
 'choices': [{'message': {'content': 'Hello! How can I help you today?'}}],
 'usage': {'prompt_tokens': 10, 'completion_tokens': 12, 'total_tokens': 22},
 'timestamp': '2024-01-15',
 'user_id': 'user123'}

Nested Dictionaries (Common in API Responses)

LLM API responses often contain nested dictionaries. Here's how to work with them:

In [141]:
```
# Typical OpenAI API response structure
```

```
openai_response = {
    "id": "chatcmpl-7X8K2vD5G1Zq3F9N4M8P1R6S",
    "object": "chat.completion",
    "created": 1686123456,
    "model": "gpt-4",
    "choices": [
        {
            "index": 0,
            "message": {
                "role": "assistant",
                "content": "I'd be happy to help you with Python programming!"
            },
            "finish_reason": "stop"
        }
    ],
    "usage": {
        "prompt_tokens": 25,
        "completion_tokens": 15,
        "total_tokens": 40
    }
}
```

Access nested data

In [142]:
```
response_text = openai_response["choices"][0]["message"]["content"]
total_tokens = openai_response["usage"]["total_tokens"]
model_used = openai_response["model"]

print(f"AI Response: {response_text}")
print(f"Tokens used: {total_tokens}")
print(f"Model: {model_used}")
```

AI Response: I'd be happy to help you with Python programming!
Tokens used: 40
Model: gpt-4

Sets: Unique Collections

Sets are collections of unique items with no duplicates. They're perfect for tracking unique users, removing duplicates, or checking membership quickly.

In [146]:
```
# Create sets
unique_users = set()
visited_pages = {"home", "chat", "settings", "profile"}
visited_pages
```

Out[146]:{'chat', 'home', 'profile', 'settings'}

In [145]:
```
all_models = ["gpt-4", "gpt-3.5", "gpt-4", "claude-3", "gpt-3.5", "gemini"]
unique_models = set(all_models)
print(f"All models: {all_models}")
print(f"Unique models: {unique_models}")
```

All models: ['gpt-4', 'gpt-3.5', 'gpt-4', 'claude-3', 'gpt-3.5', 'gemini']
Unique models: {'gpt-4', 'gemini', 'gpt-3.5', 'claude-3'}

In [147]:
```
# Add and remove items
unique_users.add("user123")
unique_users.add("user456")
unique_users.add("user123")  # Duplicate - won't be added again
print(f"Unique users: {unique_users}")
```

Unique users: {'user456', 'user123'}

In [149]:
```
# Remove items
unique_users.discard("user789")  # No error if item doesn't exist
```

In [150]:
```
unique_users.remove("user123")   # Error if item doesn't exist
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[150], line 1
----> 1 unique_users.remove("user123")   # Error if item doesn't exist

KeyError: 'user123'
```

> **Fact:** Errors are not always bad. They can be useful.
> Suppose you want to do something if you try to remove an item from the set and it's not there. You can use the KeyError messege in your logic to achieve what you want.

Useful Set Operations

In [151]:
```python
# Users who used different features
chat_users = {"alice", "bob", "charlie", "diana"}
api_users = {"bob", "charlie", "eve", "frank"}

# Union: users who used either feature
all_users = chat_users.union(api_users)
print(f"All users: {all_users}")
```

All users: {'diana', 'frank', 'eve', 'charlie', 'bob', 'alice'}

In [152]:
```python
# Intersection: users who used both features
power_users = chat_users.intersection(api_users)
print(f"Power users: {power_users}")
```

Power users: {'charlie', 'bob'}

In [153]:
```python
# Difference: users who used chat but not API
chat_only = chat_users.difference(api_users)
print(f"Chat-only users: {chat_only}")
```

Chat-only users: {'diana', 'alice'}

In [154]:
```python
# Check relationships
print(f"Are all chat users also API users? {chat_users.issubset(api_users)}")
print(f"Are there any common users? {bool(chat_users.intersection(api_users))}")
```

Are all chat users also API users? False
Are there any common users? True

3. Program Flow: Making Decisions and Repeating Actions

Real applications need to make decisions and repeat actions based on different conditions. In LLM applications, you'll constantly need to check user input, handle API responses, and process data in loops.

Conditional Statements: Making Decisions

Conditional statements let your program choose different paths based on the data it's working with. Think of them as decision trees that guide your program's behavior.

Basic if, elif, else Structure

In [155]:
```python
# Example: Choosing AI model based on user plan
user_plan = "premium"
message_length = 500

if user_plan == "premium":
    if message_length > 1000:
        recommended_model = "gpt-4-32k"
    else:
        recommended_model = "gpt-4"
elif user_plan == "pro":
    recommended_model = "gpt-3.5-turbo"
else:
    recommended_model = "gpt-3.5-turbo"
    max_tokens = 100  # Limit for free users

print(f"Recommended model: {recommended_model}")
```

Recommended model: gpt-4

Comparison Operators

In [156]:
```python
# Common comparison operators used in LLM applications
api_calls_remaining = 450
token_count = 1500
user_age = 25
error_rate = 0.02

# Equality and inequality
if api_calls_remaining == 0:
    print("No API calls remaining")
```

In [157]:
```python
if token_count != 0:
    print("Message contains tokens")
```

Message contains tokens

In [158]:
```python
# Numerical comparisons
if api_calls_remaining < 50:
    print("Warning: Low API calls remaining")
```

In [159]:
```python
if token_count > 2000:
    print("Message is quite long")
```
In [160]:
```python
if user_age >= 18:
    print("User is an adult")
```
User is an adult

In [ ]:
```python
if error_rate <= 0.05:
    print("Error rate is acceptable")
```
In [161]:
```python
# String comparisons
model_name = "gpt-4"
if model_name in ["gpt-4", "gpt-4-turbo", "gpt-4-32k"]:
    print("Using a GPT-4 variant")
```
Using a GPT-4 variant

Logical Operators: and, or, not

In [162]:
```python
# Combine multiple conditions
user_plan = "premium"
api_calls_remaining = 75
has_valid_api_key = True
message_length = 1200

# AND: both conditions must be true
if user_plan == "premium" and api_calls_remaining > 50:
    print("Can use advanced features")
```
Can use advanced features

In [163]:
```python
# OR: at least one condition must be true
if api_calls_remaining < 10 or user_plan == "free":
    print("Consider upgrading plan")
```
In [165]:
```python
# NOT: reverse the condition
if not has_valid_api_key:
    print("Please check your API key")
else:
    print("API key not valid")
```
API key not valid

In [166]:
```python
# Complex combinations
if (user_plan == "premium" or user_plan == "pro") and has_valid_api_key and message_length < 4000:
    model = "gpt-4"
    print(f"Using {model} for this request")
else:
    model = "gpt-3.5-turbo"
    print(f"Using {model} (fallback)")
```
Using gpt-4 for this request

Loops: Repeating Actions

Loops let you repeat code multiple times, which is essential for processing lists of data, handling multiple API responses, or creating interactive applications.

For Loops: Iterating Over Collections

For loops are perfect when you know what you want to iterate over:

In [ ]:
```python
# Process a list of user messages
messages = [
    "Hello, I need help with Python",
    "Can you explain loops?",
    "What about functions?",
    "Thanks for your help!"
]
```
In [169]:
```python
# Process each message
for message in messages:
    word_count = len(message.split())
    print(f"Message \"{message}\": {word_count} words")
```
Message "User: Hello": 2 words
Message "AI: Hi! How can I help you today?": 8 words
Message "User: I need help with Python": 6 words

In [170]:
```python
# Process each message
for i, message in enumerate(messages):
    word_count = len(message.split())
    print(f"Message {i+1}: {word_count} words")
```

Message 1: 2 words
Message 2: 8 words
Message 3: 6 words

In [173]:
```
# Iterate over dictionary keys and values
api_costs = {
    "gpt-4": 0.03,
    "gpt-3.5-turbo": 0.002,
    "claude-3": 0.025,
    "gemini-pro": 0.001
}

print("\nAPI Cost Analysis:")
total_cost = 0
for model, cost in api_costs.items():
    requests_per_day = 100
    daily_cost = (cost * requests_per_day)
    total_cost += daily_cost
    print(f"{model}: ${daily_cost:.2f} per day")

print(f"Total daily cost: ${total_cost:.2f}")
```

API Cost Analysis:
gpt-4: $3.00 per day
gpt-3.5-turbo: $0.20 per day
claude-3: $2.50 per day
gemini-pro: $0.10 per day
Total daily cost: $5.80

Range: Creating Number Sequences

In [174]:
```
total_messages = 1000
batch_size = 50
range(0, total_messages, batch_size)
```

Out[174]:range(0, 1000, 50)

In [176]:
```
list(range(0, total_messages, batch_size))
```

Out[176]:[0,
 50,
 100,
 150,
 200,
 250,
 300,
 350,
 400,
 450,
 500,
 550,
 600,
 650,
 700,
 750,
 800,
 850,
 900,
 950]

In [178]:
```
print("Processing messages in batches:")
for batch_start in range(0, total_messages, batch_size):
    batch_end = min(batch_start + batch_size, total_messages)
    print(f"Processing messages {batch_start+1}-{batch_end}")
    # Do Something interesting here
```

Processing messages in batches:
Processing messages 1-50
Processing messages 51-100
Processing messages 101-150
Processing messages 151-200
Processing messages 201-250
Processing messages 251-300
Processing messages 301-350
Processing messages 351-400
Processing messages 401-450
Processing messages 451-500
Processing messages 501-550
Processing messages 551-600
Processing messages 601-650
Processing messages 651-700
Processing messages 701-750
Processing messages 751-800
Processing messages 801-850
Processing messages 851-900
Processing messages 901-950
Processing messages 951-1000

While Loops: Repeating Until a Condition Changes

While loops continue until a condition becomes False. They're useful for interactive applications or when you don't know exactly how many iterations you'll need:

In [7]:
```python
# Simple chat simulation
messages = ["Hello", "How are you?", "Tell me a joke", "goodbye"]
count = 0

while count < len(messages):
    user_msg = messages[count]
    print(f"count is {count}")
    print(f"User: {user_msg}")

    if user_msg.lower() == "goodbye":
        print("AI: Goodbye!")
        break

    responses = ["Hi there!", "I'm good!", "Why did the code break? Too many bugs!"]

    print(f"AI: {responses[count]} \n")
    count += 1
```

count is 0
User: Hello
AI: Hi there!

count is 1
User: How are you?
AI: I'm good!

count is 2
User: Tell me a joke
AI: Why did the code break? Too many bugs!

count is 3
User: goodbye
AI: Goodbye!

Loop Control: break, continue, and else

In [193]:
```python
# Basic for loop with break
print("=== Break Example ===")
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    if num == 3:
        print("Found 3, stopping!")
        break
    print(f"Number: {num}")
```

=== Break Example ===
Number: 1
Number: 2
Found 3, stopping!

In [194]:
```
# Basic for loop with continue
print("=== Continue Example ===")
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    if num == 3:
        print("Skipping 3")
        continue
    print(f"Number: {num}")
```

=== Continue Example ===
Number: 1
Number: 2
Skipping 3
Number: 4
Number: 5

In [ ]:
```
# For loop with else:
print("=== For-Else Example ===")
numbers = [1, 2, 4, 5]  # No 3 in this list
for num in numbers:
    if num == 3:
        print("Found 3!")
        break
    print(f"Number: {num}")
else:
    print("No 3 found in the list")
```

=== For-Else Example ===
Number: 1
Number: 2
Number: 4
Number: 5
No 3 found in the list

In [ ]:
```
# Simple error counting
print("=== Error Counting ===")
responses = ["success", "error", "success", "error", "error"]
error_count = 0

for response in responses:
    if response == "error":
        error_count += 1
        if error_count >= 2:
            print("Too many errors, stopping")
            break
    else:
        print("Processing successful response")

print(f"Total errors: {error_count}")
```

=== Error Counting ===
Processing successful response
Processing successful response
Too many errors, stopping
Total errors: 2

Nested Loops: Processing Complex Data

In [190]:
```
# Analyze conversation data across multiple users
users_conversations = {
    "alice": [
        {"message": "Hello", "timestamp": "10:00", "tokens": 5},
        {"message": "How do I use Python?", "timestamp": "10:05", "tokens": 15},
        {"message": "Thanks!", "timestamp": "10:10", "tokens": 3}
    ],
    "bob": [
        {"message": "Hi there", "timestamp": "11:00", "tokens": 7},
        {"message": "Can you help with AI?", "timestamp": "11:02", "tokens": 18}
    ],
    "charlie": [
        {"message": "Good morning", "timestamp": "09:30", "tokens": 8},
        {"message": "I need coding help", "timestamp": "09:35", "tokens": 12},
        {"message": "Show me examples", "timestamp": "09:40", "tokens": 10},
```

```
            {"message": "Perfect, thank you", "timestamp": "09:45", "tokens": 11}
        ]
    }
```

In [191]:
```
users_conversations.items()
```

Out[191]:dict_items([('alice', [{'message': 'Hello', 'timestamp': '10:00', 'tokens': 5}, {'message': 'How do I use Python?', 'timestamp': '10:05', 'tokens': 15}, {'message': 'Thanks!', 'timestamp': '10:10', 'tokens': 3}]), ('bob', [{'message': 'Hi there', 'timestamp': '11:00', 'tokens': 7}, {'message': 'Can you help with AI?', 'timestamp': '11:02', 'tokens': 18}]), ('charlie', [{'message': 'Good morning', 'timestamp': '09:30', 'tokens': 8}, {'message': 'I need coding help', 'timestamp': '09:35', 'tokens': 12}, {'message': 'Show me examples', 'timestamp': '09:40', 'tokens': 10}, {'message': 'Perfect, thank you', 'timestamp': '09:45', 'tokens': 11}])])

In [192]:
```
# Analyze usage patterns
print("User Activity Analysis:")
print("=" * 50)

total_messages = 0
total_tokens = 0

for username, conversations in users_conversations.items():
    user_messages = len(conversations)
    user_tokens = sum(msg["tokens"] for msg in conversations)

    print(f"\n👤 {username.title()}:")
    print(f"   Messages: {user_messages}")
    print(f"   Tokens used: {user_tokens}")

    # Find peak activity time
    message_times = [msg["timestamp"] for msg in conversations]
    print(f"   Active between: {min(message_times)} - {max(message_times)}")

    # Check for long messages
    long_messages = [msg for msg in conversations if msg["tokens"] > 10]
    if long_messages:
        print(f"   Long messages ({len(long_messages)}): ", end="")
        print(", ".join([f"'{msg['message'][:20]}...'" for msg in long_messages]))

    total_messages += user_messages
    total_tokens += user_tokens

print(f"\n📊 Overall Statistics:")
print(f"   Total users: {len(users_conversations)}")
print(f"   Total messages: {total_messages}")
print(f"   Total tokens: {total_tokens}")
print(f"   Average messages per user: {total_messages / len(users_conversations):.1f}")
print(f"   Average tokens per message: {total_tokens / total_messages:.1f}")
```

```
User Activity Analysis:
==================================================

👤 Alice:
   Messages: 3
   Tokens used: 23
   Active between: 10:00 - 10:10
   Long messages (1): 'How do I use Python?...'

👤 Bob:
   Messages: 2
   Tokens used: 25
   Active between: 11:00 - 11:02
   Long messages (1): 'Can you help with AI...'

👤 Charlie:
   Messages: 4
   Tokens used: 41
   Active between: 09:30 - 09:45
   Long messages (2): 'I need coding help...', 'Perfect, thank you...'

📊 Overall Statistics:
   Total users: 3
   Total messages: 9
   Total tokens: 89
   Average messages per user: 3.0
   Average tokens per message: 9.9
```

4. Functions: Creating Reusable Code

Functions are the building blocks of well-organized code. In LLM applications, you'll use functions to handle API calls, process responses, validate input, and organize your logic into manageable pieces. Think of functions as recipes that take ingredients (inputs) and produce a dish (output).

Why Functions Matter for LLM Applications

- **Reusability**: Write once, use many times
- **Organization**: Keep related code together
- **Testing**: Easier to test small pieces of functionality
- **Maintenance**: Easier to update and fix bugs
- **Collaboration**: Other developers can understand and use your functions

Basic Function Definition

In [198]:
```python
def greet_user(name):
    """
    Generate a personalized greeting for a user.

    Args:
        name (str): The user's name

    Returns:
        str: A personalized greeting message
    """
    return f"Hello, {name}! Welcome to our AI assistant!"

# Using the function
welcome_message = greet_user("Dr. Smith")
print(welcome_message)  # "Hello, Dr. Smith! Welcome to our AI assistant!"
```

Hello, Dr. Smith! Welcome to our AI assistant!

In [199]:
```python
# Functions can be called multiple times
for user in ["Alice", "Bob", "Charlie"]:
    print(greet_user(user))
```

Hello, Alice! Welcome to our AI assistant!
Hello, Bob! Welcome to our AI assistant!
Hello, Charlie! Welcome to our AI assistant!

In [201]:
```python
### Functions with Multiple Parameters

def create_ai_prompt(user_message, system_role="helpful assistant", temperature=0.7):
    """
    Create a structured prompt for an AI model.

    Args:
        user_message (str): The user's input message
        system_role (str): The AI's role/personality (default: "helpful assistant")
        temperature (float): Response creativity level (default: 0.7)

    Returns:
        dict: Structured prompt data
    """
    prompt_data = {
        "system": f"You are a {system_role}.",
        "user": user_message,
        "temperature": temperature,
        "timestamp": "2024-01-15 10:30:00"  # In real app, use datetime
    }

    return prompt_data

# Using the function with different parameters
basic_prompt = create_ai_prompt("How do I learn Python?")
basic_prompt
```

Out[201]:
```
{'system': 'You are a helpful assistant.',
 'user': 'How do I learn Python?',
 'temperature': 0.7,
 'timestamp': '2024-01-15 10:30:00'}
```

In [202]:
```python
coding_prompt = create_ai_prompt(
    "Explain object-oriented programming",
    system_role="expert Python instructor",
```

```
    temperature=0.3
)
coding_prompt
```

Out[202]:{'system': 'You are a expert Python instructor.',
'user': 'Explain object-oriented programming',
'temperature': 0.3,
'timestamp': '2024-01-15 10:30:00'}

Default Parameters and Keyword Arguments

Default parameters make functions more flexible and easier to use:

In [206]:
```
def send_api_request(message, model="gpt-3.5-turbo", max_tokens=1000, temperature=0.7, stream=False)
    """
    Simulate sending a request to an AI API.

    Args:
        message (str): User's message
        model (str): AI model to use (default: "gpt-3.5-turbo")
        max_tokens (int): Maximum response length (default: 1000)
        temperature (float): Response creativity (default: 0.7)
        stream (bool): Whether to stream response (default: False)

    Returns:
        dict: Simulated API response
    """

    # Simulate API call
    response = {
        "model": model,
        "message": message,
        "settings": {
            "max_tokens": max_tokens,
            "temperature": temperature,
            "stream": stream
        },
        "response": f"This is a simulated response to: '{message}'"
    }

    return response
```

In [207]:
```
response1 = send_api_request("Hello, how are you?")
response1
```

Out[207]:{'model': 'gpt-3.5-turbo',
'message': 'Hello, how are you?',
'settings': {'max_tokens': 1000, 'temperature': 0.7, 'stream': False},
'response': "This is a simulated response to: 'Hello, how are you?'"}

In [208]:
```
response2 = send_api_request("Explain quantum physics", model="gpt-4")
response2
```

Out[208]:{'model': 'gpt-4',
'message': 'Explain quantum physics',
'settings': {'max_tokens': 1000, 'temperature': 0.7, 'stream': False},
'response': "This is a simulated response to: 'Explain quantum physics'"}

In [209]:
```
response3 = send_api_request(
    message="Write a poem about programming",
    temperature=0.9,  # More creative
    max_tokens=500,
    stream=True
)
response3
```

Out[209]:{'model': 'gpt-3.5-turbo',
'message': 'Write a poem about programming',
'settings': {'max_tokens': 500, 'temperature': 0.9, 'stream': True},
'response': "This is a simulated response to: 'Write a poem about programming'"}

In [210]:
```
response4 = send_api_request("What's the weather?", "gpt-4", temperature=0.2)
response4
```

Out[210]:{'model': 'gpt-4',
'message': "What's the weather?",
'settings': {'max_tokens': 1000, 'temperature': 0.2, 'stream': False},
'response': "This is a simulated response to: 'What's the weather?'"}

Variable Arguments: *args and* *kwargs

Sometimes you don't know exactly how many arguments a function will receive. Python provides `*args` for variable positional arguments and `**kwargs` for variable keyword arguments:

Using `*args`

In [218]:
```
def calculate_total_tokens(*token_counts):
    """
    Calculate total tokens from multiple API calls.

    Args:
        *token_counts: Variable number of token count integers

    Returns:
        int: Total number of tokens
    """
    total = sum(token_counts)

    return total
```

In [220]:
```
total1 = calculate_total_tokens(150, 200, 75)  # 3 API calls
print(f" API used {total1} tokens")
```

API used 425 tokens

In [221]:
```
total2 = calculate_total_tokens(100, 250, 180, 90, 300)  # 5 API calls
print(f" API used {total2} tokens")
```

API used 920 tokens

In [222]:
```
total3 = calculate_total_tokens(500)  # 1 API call
print(f" API used {total3} tokens")
```

API used 500 tokens

In [223]:
```
token_list = [120, 340, 280, 150]
total4 = calculate_total_tokens(*token_list)
print(f" API used {total4} tokens")
```

API used 890 tokens

In [224]:
```
#### Using `**kwargs`
def create_model_config(model_name, **additional_settings):
    """
    Create a configuration dictionary for an AI model.

    Args:
        model_name (str): Name of the AI model
        **additional_settings: Any additional configuration options

    Returns:
        dict: Complete model configuration
    """
    # Start with basic configuration
    config = {
        "model": model_name,
        "version": "latest",
        "created_at": "2024-01-15"
    }

    # Add all additional settings
    config.update(additional_settings)

    print(f"Created config for {model_name} with {len(additional_settings)} additional settings")
    return config

# Using with different keyword arguments
config1 = create_model_config("gpt-4", temperature=0.7, max_tokens=2000)

config1
```

Created config for gpt-4 with 2 additional settings

Out[224]:
```
{'model': 'gpt-4',
 'version': 'latest',
 'created_at': '2024-01-15',
 'temperature': 0.7,
 'max_tokens': 2000}
```

In [226]:
```
config2 = create_model_config(
    "claude-3",
```

```
        temperature=0.8,
        max_tokens=4000,
        top_p=0.9,
        frequency_penalty=0.1,
        presence_penalty=0.1
    )
    config2
```

Created config for claude-3 with 5 additional settings

Out[226]:{'model': 'claude-3',
 'version': 'latest',
 'created_at': '2024-01-15',
 'temperature': 0.8,
 'max_tokens': 4000,
 'top_p': 0.9,
 'frequency_penalty': 0.1,
 'presence_penalty': 0.1}

In [227]:
```
config3 = create_model_config("gemini-pro", stream=True, safety_settings="high")
config3
```

Created config for gemini-pro with 2 additional settings

Out[227]:{'model': 'gemini-pro',
 'version': 'latest',
 'created_at': '2024-01-15',
 'stream': True,
 'safety_settings': 'high'}

In [228]:
```
#### Combining `*args` and `**kwargs`
```

In [232]:
```
def flexible_api_call(endpoint, *data_items, **options):
    """
    A flexible function that can handle various API call patterns.

    Args:
        endpoint (str): API endpoint URL
        *data_items: Variable number of data items to send
        **options: Variable keyword arguments for API options

    Returns:
        dict: Simulated API response
    """

    print(f"Calling endpoint: {endpoint}")
    print(f"Data items: {data_items}")
    print(f"Options: {options}")

    # Simulate API response
    response = {
        "endpoint": endpoint,
        "data_count": len(data_items),
        "options_count": len(options),
        "status": "success"
    }

    return response
response1 = flexible_api_call("/chat", "Hello", "How are you?", model="gpt-4", temperature=0.7)
```

Calling endpoint: /chat
Data items: ('Hello', 'How are you?')
Options: {'model': 'gpt-4', 'temperature': 0.7}

In [233]:
```
response2 = flexible_api_call(
    "/completion",
    "Write a story about AI",
    max_tokens=1000,
    temperature=0.9,
    top_p=0.95
)
```

Calling endpoint: /completion
Data items: ('Write a story about AI',)
Options: {'max_tokens': 1000, 'temperature': 0.9, 'top_p': 0.95}

In [236]:
```
response3 = flexible_api_call("/translate", "Bonjour le monde", "Salut le monde", source="fr", targe
```

Calling endpoint: /translate
Data items: ('Bonjour le monde', 'Salut le monde')
Options: {'source': 'fr', 'target': 'en'}

Type Hints: Making Your Code More Reliable

Type hints are a modern Python feature that specify what types of data your functions expect and return. They make your code more readable, help catch errors early, and work great with AI coding assistants!

Basic Type Hints

In [238]:
```python
from typing import List, Dict, Optional, Union

def calculate_api_cost(token_count: int, cost_per_1k: float) -> float:
    """
    Calculate the cost of an API call based on token usage.

    Args:
        token_count: Number of tokens used
        cost_per_1k: Cost per 1000 tokens

    Returns:
        Total cost in dollars
    """
    return (token_count / 1000) * cost_per_1k

def process_user_messages(messages: List[str]) -> Dict[str, int]:
    """
    Process a list of user messages and return statistics.

    Args:
        messages: List of user message strings

    Returns:
        Dictionary with message statistics
    """
    stats = {
        "total_messages": len(messages),
        "total_words": sum(len(msg.split()) for msg in messages),
        "total_characters": sum(len(msg) for msg in messages)
    }

    return stats
```
In [239]:
```python
# Using the functions
cost = calculate_api_cost(1500, 0.002)  # 1500 tokens at $0.002 per 1k
print(f"API cost: ${cost:.4f}")

user_msgs = ["Hello there", "How can I learn Python?", "Thanks for the help!"]
message_stats = process_user_messages(user_msgs)
print("Message stats:", message_stats)
```
API cost: $0.0030
Message stats: {'total_messages': 3, 'total_words': 11, 'total_characters': 54}

In [ ]:
```
### Type Hints: A Modern Python Feature for Better Code

**Type hints** are a relatively new Python feature (introduced in Python 3.5+) that you may not have e

#### Why Type Hints Matter for LLM Applications

- **Catch errors early**: Know if you're passing the wrong type of data before running your code
- **Better documentation**: Makes it clear what your functions expect
- **AI assistant friendly**: Tools like GitHub Copilot understand type hints and give better suggestio
- **Professional standard**: Modern Python development increasingly uses type hints

#### Basic Type Hints
```
In [240]:
```python
# Without type hints (old way)
def calculate_cost_1(tokens, rate):
    return (tokens / 1000) * rate
```
In [241]:
```python
# With type hints (modern way)
def calculate_cost_2(token_count: int, cost_per_1k: float) -> float:
    """
    Calculate the cost of an API call based on token usage.

    Args:
        token_count: Number of tokens used (must be an integer)
        cost_per_1k: Cost per 1000 tokens (must be a float)
```

```
    Returns:
        Total cost in dollars (returns a float)
    """
    return (token_count / 1000) * cost_per_1k
```

In [243]:
```
# The type hints tell us exactly what to expect
cost = calculate_cost_1(1500, 0.002)  # Clear: integer and float
print(f"API cost: ${cost:.4f}")
```

API cost: $0.0030

In [244]:
```
# The type hints tell us exactly what to expect
cost = calculate_cost_2(1500, 0.002)  # Clear: integer and float
print(f"API cost: ${cost:.4f}")
```

API cost: $0.0030

Type Hints for Collections

For lists, dictionaries, and other collections, you need to import from the `typing` module:

In [246]:
```
from typing import List, Dict

def count_words(messages: List[str]) -> Dict[str, int]:
    """
    Count total words and messages.

    Args:
        messages: A list of strings

    Returns:
        A dictionary with statistics
    """
    return {
        "total_messages": len(messages),
        "total_words": sum(len(msg.split()) for msg in messages)
    }

# Usage is the same, but now it's clear what types are expected
user_messages = ["Hello", "How are you?", "Goodbye"]
stats = count_words(user_messages)
print(stats)
```

{'total_messages': 3, 'total_words': 5}

Optional Values

Use `Optional` when a parameter or return value might be `None`:

In [249]:
```
from typing import Optional

def get_user_setting(user_id: str, setting: str) -> Optional[str]:
    """
    Get a user setting, or None if not found.

    Args:
        user_id: User identifier
        setting: Setting name to look up

    Returns:
        Setting value, or None if not found
    """
    user_settings = {
        "alice": {"theme": "dark", "model": "gpt-4"}
    }

    user_data = user_settings.get(user_id)
    if user_data:
        return user_data.get(setting, None)

# Clear that this might return None
theme = get_user_setting("alice", "theme")  # Returns "dark"
theme
```

Out[249]:'dark'

In [250]:
```
missing = get_user_setting("bob", "theme")   # Returns None
print(missing)
```

None

The Key Point

Type hints don't change how Python runs—they're like comments that help you and others understand your code better. You can still run the code without them, but they make your programs more professional and easier to debug.

Compare these two function definitions:

```
# Hard to understand what this expects
def process(data, options, callback):
    pass

# Crystal clear what this expects
def process_api_response(
    data: Dict[str, Any],
    options: List[str],
    callback: Optional[Callable]
) -> bool:
    pass
```

As you build LLM applications, type hints will help you catch mistakes early and make your code more maintainable.

Lambda Functions: Quick Anonymous Functions (Bonus Knowledge)

Lambda functions are small, one-line functions perfect for simple operations. They're especially useful with functions like `map()` , `filter()` , and `sort()` :

```
# Regular function
def calculate_tokens(text):
    return len(text.split())

# Equivalent lambda function
calculate_tokens_lambda = lambda text: len(text.split())
```

In [252]:
```
def calculate_tokens(text):
    return len(text.split())

# Equivalent lambda function
calculate_tokens_lambda = lambda text: len(text.split())
```

In [253]:
```
# Both work the same way
message = "Hello, how can I help you today?"
print(f"Tokens (regular): {calculate_tokens(message)}")
print(f"Tokens (lambda): {calculate_tokens_lambda(message)}")
```

Tokens (regular): 7
Tokens (lambda): 7

In [260]:
```
# Using lambda with sort() for custom sorting
api_responses = [
    {"model": "gpt-4", "tokens": 150, "cost": 0.03},
    {"model": "gpt-3.5", "tokens": 200, "cost": 0.02},
    {"model": "claude-3", "tokens": 120, "cost": 0.025}
]

# Sort by cost (ascending)
sorted_by_cost = sorted(api_responses, key=lambda x: x["cost"])
print("Sorted by cost:", sorted_by_cost)


# Sort by tokens (descending)
sorted_by_tokens = sorted(api_responses, key=lambda x: x["tokens"], reverse=True)
print("Sorted by tokens:", sorted_by_tokens)
```

Sorted by cost: [{'model': 'gpt-3.5', 'tokens': 200, 'cost': 0.02}, {'model': 'claude-3', 'tokens': 120, 'cost': 0.025}, {'model': 'gpt-4', 'tokens': 150, 'cost': 0.03}]
Sorted by tokens: [{'model': 'gpt-3.5', 'tokens': 200, 'cost': 0.02}, {'model': 'gpt-4', 'tokens': 150, 'cost': 0.03}, {'model': 'claude-3', 'tokens': 120, 'cost': 0.025}]

Higher-Order Functions: Functions That Work with Other Functions

Higher-order functions take other functions as arguments or return functions. They're powerful tools for processing data:

`map()` and `filter()`

In [261]:
```
from functools import reduce

# Sample conversation data
conversation_data = [
```

```
    {"user": "alice", "message": "Hello there!", "sentiment": "positive", "tokens": 10},
    {"user": "bob", "message": "I'm frustrated with this", "sentiment": "negative", "tokens": 15},
    {"user": "charlie", "message": "This is amazing!", "sentiment": "positive", "tokens": 12},
    {"user": "diana", "message": "Could be better", "sentiment": "neutral", "tokens": 8},
    {"user": "eve", "message": "Love this feature!", "sentiment": "positive", "tokens": 11}
]
```

In [262]:
```
# map(): Transform each item
print("=== Using map() ===")
# Extract just the token counts
token_counts = map(lambda x: x["tokens"], conversation_data)
print(f"Token counts: {list(token_counts)}")
```

=== Using map() ===
Token counts: [10, 15, 12, 8, 11]

In [263]:
```
# Create summary strings
summaries = map(
    lambda x: f"{x['user']}: {x['tokens']} tokens ({x['sentiment']})",
    conversation_data
)
for summary in summaries:
    print(summary)
```

alice: 10 tokens (positive)
bob: 15 tokens (negative)
charlie: 12 tokens (positive)
diana: 8 tokens (neutral)
eve: 11 tokens (positive)

In [266]:
```
conversation_data
```

Out[266]:
```
[{'user': 'alice',
  'message': 'Hello there!',
  'sentiment': 'positive',
  'tokens': 10},
 {'user': 'bob',
  'message': "I'm frustrated with this",
  'sentiment': 'negative',
  'tokens': 15},
 {'user': 'charlie',
  'message': 'This is amazing!',
  'sentiment': 'positive',
  'tokens': 12},
 {'user': 'diana',
  'message': 'Could be better',
  'sentiment': 'neutral',
  'tokens': 8},
 {'user': 'eve',
  'message': 'Love this feature!',
  'sentiment': 'positive',
  'tokens': 11}]
```

In [265]:
```
# filter(): Keep only items that meet a condition
print("\n=== Using filter() ===")
# Get only positive messages
positive_messages = filter(lambda x: x["sentiment"] == "positive", conversation_data)
for msg in positive_messages:
    print(f"  {msg['user']}: {msg['message']}")
```

=== Using filter() ===
  alice: Hello there!
  charlie: This is amazing!
  eve: Love this feature!

In [267]:
```
# Get high-token messages
high_token_messages = filter(lambda x: x["tokens"] > 10, conversation_data)
print(f"\nHigh-token messages: {len(list(high_token_messages))}")
```

High-token messages: 3

Conclusion

You've now covered all the essential Python concepts needed to build LLM applications:

Key Takeaways

1. **Data Types**: Understanding strings, numbers, booleans, and None helps you handle user input and API responses correctly.

2. **Data Structures**: Lists for sequences, dictionaries for key-value data (like JSON), sets for unique collections, and tuples for immutable data.

3. **Program Flow**: Conditional statements for decision-making and loops for processing multiple items.

4. **Functions**: The building blocks of clean, reusable code. Use type hints to make your code more reliable and AI-assistant-friendly.

Best Practices Recap

1. **Use type hints** - They make your code more reliable and easier to work with
2. **Write clear function names** - `process_user_message()` is better than `process()`
3. **Handle errors gracefully** - Always validate user input and API responses
4. **Keep functions focused** - Each function should do one thing well
5. **Use meaningful variable names** - `user_message` is better than `msg`