OpenAI API Tutorial: Complete Guide to GPT Models and Beyond

Table of Contents

Introduction to OpenAI API

The OpenAI API provides access to powerful AI models including GPT-4, GPT-3.5, DALL-E, Whisper, and more. This tutorial covers everything you need to know to integrate OpenAI's models into your applications.

Available Models:

- **GPT-4**: Most capable model for complex reasoning
- **GPT-3.5**: Fast and efficient for most tasks
- **GPT-4 Vision**: Understands images and text
- **DALL-E 3**: Generate images from text
- **Whisper**: Speech-to-text transcription
- **TTS**: Text-to-speech synthesis

Setup and Authentication

Installation

```
# Install the OpenAI Python library
pip install openai

# For async support
pip install openai[async]

# Latest version with all features
pip install openai>=1.0.0
```

API Key Setup

```
import openai
import os
from openai import OpenAI

# Method 1: Environment variable (recommended)
os.environ["OPENAI_API_KEY"] = "your-api-key-here"
client = OpenAI()

# Method 2: Direct initialization
client = OpenAI(api_key="your-api-key-here")

# Method 3: Using Azure OpenAI
from openai import AzureOpenAI
azure_client = AzureOpenAI(
    api_key="your-azure-key",
    api_version="2023-12-01-preview",
    azure_endpoint="https://your-endpoint.openai.azure.com/"
)
```

Basic Configuration

```
# Set default parameters
client = OpenAI(
    api_key="your-key",
    organization="your-org-id",  # Optional
    project="your-project-id",   # Optional
    base_url="https://api.openai.com/v1",  # Custom endpoint if needed
    default_headers={"Custom-Header": "value"}
)
```

Chat Completions

Basic Chat Completion

```
def basic_chat_completion():
    response = client.chat.completions.create(
        model="gpt-4",
        messages=[
            {"role": "system", "content": "You are a helpful assistant."},
            {"role": "user", "content": "Hello! How can you help me today?"}
        ],
        max_tokens=150,
        temperature=0.7
    )

    return response.choices[0].message.content

print(basic_chat_completion())
```

Advanced Parameters

```python
def advanced_chat_completion():
    response = client.chat.completions.create(
        model="gpt-4",
        messages=[
            {"role": "system", "content": "You are an expert Python developer."},
            {"role": "user", "content": "Explain list comprehensions with examples."}
        ],
        max_tokens=500,
        temperature=0.3,           # Lower = more focused
        top_p=0.9,                # Nucleus sampling
        frequency_penalty=0.0,     # Reduce repetition
        presence_penalty=0.0,      # Encourage new topics
        stop=["\n\n", "###"],      # Stop sequences
        seed=42                    # For reproducible outputs
    )

    return response.choices[0].message.content
```

Streaming Responses

```python
def streaming_chat():
    stream = client.chat.completions.create(
        model="gpt-4",
        messages=[{"role": "user", "content": "Tell me a long story about AI"}],
        stream=True
    )

    full_response = ""
    for chunk in stream:
        if chunk.choices[0].delta.content is not None:
            content = chunk.choices[0].delta.content
            print(content, end="", flush=True)
            full_response += content

    return full_response
```

Conversation Management

```
class ChatManager:
    def __init__(self, system_message="You are a helpful assistant."):
        self.messages = [{"role": "system", "content": system_message}]
        self.client = OpenAI()

    def add_user_message(self, content):
        self.messages.append({"role": "user", "content": content})

    def add_assistant_message(self, content):
        self.messages.append({"role": "assistant", "content": content})

    def get_response(self, user_input):
        self.add_user_message(user_input)

        response = self.client.chat.completions.create(
            model="gpt-4",
            messages=self.messages,
            max_tokens=500,
            temperature=0.7
        )

        assistant_message = response.choices[0].message.content
        self.add_assistant_message(assistant_message)

        return assistant_message

    def clear_history(self, keep_system=True):
        if keep_system and self.messages[0]["role"] == "system":
            self.messages = [self.messages[0]]
        else:
            self.messages = []

# Usage
chat = ChatManager()
response1 = chat.get_response("What is machine learning?")
response2 = chat.get_response("Can you give me an example?")
```

Text Generation

Legacy Completions (GPT-3.5-turbo-instruct)

```
def text_completion():
    response = client.completions.create(
        model="gpt-3.5-turbo-instruct",
        prompt="Complete this sentence: The future of AI is",
        max_tokens=100,
        temperature=0.8,
        stop=["\n"]
    )

    return response.choices[0].text.strip()
```

Creative Writing Assistant

```
def creative_writer(prompt, style="narrative", length="medium"):
    length_map = {
        "short": 200,
        "medium": 500,
        "long": 1000
    }

    system_message = f"""You are a creative writing assistant specializing in {style} writing.
    Create engaging, well-structured content that captures the reader's attention."""

    response = client.chat.completions.create(
        model="gpt-4",
        messages=[
            {"role": "system", "content": system_message},
            {"role": "user", "content": f"Write a {style} piece based on: {prompt}"}
        ],
        max_tokens=length_map.get(length, 500),
        temperature=0.8,
        presence_penalty=0.1
    )

    return response.choices[0].message.content
```

Function Calling

Basic Function Calling

```
import json

def get_weather(location, unit="celsius"):
    """Simulate weather API call"""
    return {
        "location": location,
        "temperature": "22",
        "unit": unit,
        "condition": "sunny"
    }

def function_calling_example():
    # Define the function schema
    tools = [
        {
            "type": "function",
            "function": {
                "name": "get_weather",
                "description": "Get current weather for a location",
                "parameters": {
                    "type": "object",
                    "properties": {
                        "location": {
                            "type": "string",
                            "description": "City name"
                        },
                        "unit": {
                            "type": "string",
                            "enum": ["celsius", "fahrenheit"]
                        }
                    },
                    "required": ["location"]
                }
            }
        }
    ]

    messages = [
        {"role": "user", "content": "What's the weather like in Paris?"}
    ]

    # First call - model decides to use function
    response = client.chat.completions.create(
        model="gpt-4",
        messages=messages,
        tools=tools,
        tool_choice="auto"
    )

    # Check if model wants to call a function
    if response.choices[0].message.tool_calls:
        tool_call = response.choices[0].message.tool_calls[0]
        function_name = tool_call.function.name
        function_args = json.loads(tool_call.function.arguments)
```

```
    # Execute the function
    if function_name == "get_weather":
        function_result = get_weather(**function_args)

        # Add function result to conversation
        messages.append(response.choices[0].message)
        messages.append({
            "role": "tool",
            "content": json.dumps(function_result),
            "tool_call_id": tool_call.id
        })

        # Get final response
        final_response = client.chat.completions.create(
            model="gpt-4",
            messages=messages
        )

        return final_response.choices[0].message.content

return response.choices[0].message.content
```

Multiple Function Agent

```
class FunctionAgent:
    def __init__(self):
        self.client = OpenAI()
        self.functions = {
            "calculator": self.calculate,
            "search": self.search,
            "save_note": self.save_note
        }

        self.tools = [
            {
                "type": "function",
                "function": {
                    "name": "calculator",
                    "description": "Perform mathematical calculations",
                    "parameters": {
                        "type": "object",
                        "properties": {
                            "expression": {"type": "string", "description": "Math expression"}
                        },
                        "required": ["expression"]
                    }
                }
            },
            {
                "type": "function",
                "function": {
                    "name": "search",
                    "description": "Search for information",
                    "parameters": {
                        "type": "object",
                        "properties": {
                            "query": {"type": "string", "description": "Search query"}
                        },
                        "required": ["query"]
                    }
                }
            }
        ]

    def calculate(self, expression):
        try:
            result = eval(expression)  # Use safely in production!
            return f"Result: {result}"
        except:
            return "Error in calculation"

    def search(self, query):
        return f"Search results for '{query}': [Simulated results]"

    def save_note(self, content):
        # Simulate saving
        return f"Note saved: {content[:50]}..."
```

```
    def run(self, user_input):
        messages = [{"role": "user", "content": user_input}]

        response = self.client.chat.completions.create(
            model="gpt-4",
            messages=messages,
            tools=self.tools,
            tool_choice="auto"
        )

        # Handle tool calls
        if response.choices[0].message.tool_calls:
            messages.append(response.choices[0].message)

            for tool_call in response.choices[0].message.tool_calls:
                function_name = tool_call.function.name
                function_args = json.loads(tool_call.function.arguments)

                if function_name in self.functions:
                    result = self.functions[function_name](**function_args)
                    messages.append({
                        "role": "tool",
                        "content": result,
                        "tool_call_id": tool_call.id
                    })

            # Get final response
            final_response = self.client.chat.completions.create(
                model="gpt-4",
                messages=messages
            )
            return final_response.choices[0].message.content

        return response.choices[0].message.content

# Usage
agent = FunctionAgent()
result = agent.run("What's 25 * 47 + 123?")
```

Embeddings

Basic Embeddings

```
def get_embeddings(texts):
    if isinstance(texts, str):
        texts = [texts]

    response = client.embeddings.create(
        model="text-embedding-3-large",  # or text-embedding-3-small
        input=texts
    )

    return [embedding.embedding for embedding in response.data]

# Usage
text = "This is a sample sentence for embedding."
embeddings = get_embeddings(text)
print(f"Embedding dimension: {len(embeddings[0])}")
```

Semantic Search System

```
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity

class SemanticSearch:
    def __init__(self):
        self.client = OpenAI()
        self.documents = []
        self.embeddings = []

    def add_documents(self, docs):
        """Add documents to search index"""
        self.documents.extend(docs)

        # Get embeddings for new documents
        response = self.client.embeddings.create(
            model="text-embedding-3-large",
            input=docs
        )

        new_embeddings = [emb.embedding for emb in response.data]
        self.embeddings.extend(new_embeddings)

    def search(self, query, top_k=5):
        """Search for similar documents"""
        if not self.embeddings:
            return []

        # Get query embedding
        query_response = self.client.embeddings.create(
            model="text-embedding-3-large",
            input=[query]
        )
        query_embedding = query_response.data[0].embedding

        # Calculate similarities
        similarities = cosine_similarity(
            [query_embedding],
            self.embeddings
        )[0]

        # Get top results
        top_indices = np.argsort(similarities)[::-1][:top_k]

        results = []
        for idx in top_indices:
            results.append({
                "document": self.documents[idx],
                "similarity": similarities[idx]
            })

        return results

# Usage
search_engine = SemanticSearch()
```

```
search_engine.add_documents([
    "Python is a programming language",
    "Machine learning is a subset of AI",
    "Deep learning uses neural networks",
    "Natural language processing handles text"
])

results = search_engine.search("What is AI?", top_k=2)
for result in results:
    print(f"Similarity: {result['similarity']:.3f}")
    print(f"Document: {result['document']}\n")
```

Vision Models

Image Analysis

```
import base64
import requests

def encode_image(image_path):
    """Encode image to base64"""
    with open(image_path, "rb") as image_file:
        return base64.b64encode(image_file.read()).decode('utf-8')

def analyze_image(image_path, prompt="What's in this image?"):
    base64_image = encode_image(image_path)

    response = client.chat.completions.create(
        model="gpt-4-vision-preview",
        messages=[
            {
                "role": "user",
                "content": [
                    {"type": "text", "text": prompt},
                    {
                        "type": "image_url",
                        "image_url": {
                            "url": f"{base64_image}",
                            "detail": "high"  # or "low" for faster processing
                        }
                    }
                ]
            }
        ],
        max_tokens=300
    )

    return response.choices[0].message.content

# Usage with URL
def analyze_image_url(image_url, prompt="Describe this image"):
    response = client.chat.completions.create(
        model="gpt-4-vision-preview",
        messages=[
            {
                "role": "user",
                "content": [
                    {"type": "text", "text": prompt},
                    {"type": "image_url", "image_url": {"url": image_url}}
                ]
            }
        ],
        max_tokens=300
    )

    return response.choices[0].message.content
```

Document OCR and Analysis

```
def document_analyzer(image_path):
    """Extract and analyze text from documents"""
    base64_image = encode_image(image_path)

    response = client.chat.completions.create(
        model="gpt-4-vision-preview",
        messages=[
            {
                "role": "user",
                "content": [
                    {
                        "type": "text",
                        "text": "Extract all text from this document and provide a summary of its key points."
                    },
                    {
                        "type": "image_url",
                        "image_url": {"url": f"{base64_image}"}
                    }
                ]
            }
        ],
        max_tokens=1000
    )

    return response.choices[0].message.content
```

Audio and Speech

Speech-to-Text (Whisper)

```
def transcribe_audio(audio_file_path):
    """Transcribe audio file using Whisper"""
    with open(audio_file_path, "rb") as audio_file:
        transcription = client.audio.transcriptions.create(
            model="whisper-1",
            file=audio_file,
            response_format="text"
        )

    return transcription

def transcribe_with_timestamps(audio_file_path):
    """Get transcription with timestamps"""
    with open(audio_file_path, "rb") as audio_file:
        transcription = client.audio.transcriptions.create(
            model="whisper-1",
            file=audio_file,
            response_format="verbose_json",
            timestamp_granularities=["word"]
        )

    return transcription

# Translation
def translate_audio(audio_file_path):
    """Translate foreign language audio to English"""
    with open(audio_file_path, "rb") as audio_file:
        translation = client.audio.translations.create(
            model="whisper-1",
            file=audio_file
        )

    return translation.text
```

Text-to-Speech

```
def text_to_speech(text, voice="alloy", output_file="speech.mp3"):
    """Convert text to speech"""
    response = client.audio.speech.create(
        model="tts-1",  # or "tts-1-hd" for higher quality
        voice=voice,    # alloy, echo, fable, onyx, nova, shimmer
        input=text,
        speed=1.0       # 0.25 to 4.0
    )

    with open(output_file, "wb") as f:
        for chunk in response.iter_bytes():
            f.write(chunk)

    return output_file

# Real-time streaming
def streaming_text_to_speech(text, voice="alloy"):
    """Stream audio in real-time"""
    response = client.audio.speech.create(
        model="tts-1",
        voice=voice,
        input=text,
        response_format="opus"  # Better for streaming
    )

    # Play audio chunks as they arrive
    for chunk in response.iter_bytes(chunk_size=1024):
        # Send to audio player
        yield chunk
```

Fine-tuning

Prepare Training Data

```
import json

def prepare_training_data(examples):
    """Prepare data for fine-tuning"""
    training_data = []

    for example in examples:
        training_data.append({
            "messages": [
                {"role": "system", "content": "You are a helpful assistant."},
                {"role": "user", "content": example["input"]},
                {"role": "assistant", "content": example["output"]}
            ]
        })

    # Save to JSONL file
    with open("training_data.jsonl", "w") as f:
        for item in training_data:
            f.write(json.dumps(item) + "\n")

    return "training_data.jsonl"

# Example data
examples = [
    {"input": "What is Python?", "output": "Python is a programming language..."},
    {"input": "How do lists work?", "output": "Lists in Python are ordered collections..."}
]

training_file = prepare_training_data(examples)
```

Fine-tuning Process

```python
def create_fine_tuning_job(training_file, model="gpt-3.5-turbo"):
    """Create a fine-tuning job"""
    # Upload training file
    with open(training_file, "rb") as f:
        file_response = client.files.create(
            file=f,
            purpose="fine-tune"
        )

    # Create fine-tuning job
    job = client.fine_tuning.jobs.create(
        training_file=file_response.id,
        model=model,
        hyperparameters={
            "n_epochs": 3,
            "batch_size": 1,
            "learning_rate_multiplier": 2
        }
    )

    return job

def monitor_fine_tuning(job_id):
    """Monitor fine-tuning progress"""
    job = client.fine_tuning.jobs.retrieve(job_id)

    print(f"Job ID: {job.id}")
    print(f"Status: {job.status}")
    print(f"Model: {job.fine_tuned_model}")

    # Get events
    events = client.fine_tuning.jobs.list_events(job_id)
    for event in events.data[:5]:  # Show last 5 events
        print(f"{event.created_at}: {event.message}")

    return job

def use_fine_tuned_model(model_id, prompt):
    """Use your fine-tuned model"""
    response = client.chat.completions.create(
        model=model_id,
        messages=[
            {"role": "user", "content": prompt}
        ]
    )

    return response.choices[0].message.content
```

Best Practices

Error Handling

```python
from openai import RateLimitError, APIError
import time

def robust_api_call(func, max_retries=3, backoff_factor=2):
    """Robust API call with retry logic"""
    for attempt in range(max_retries):
        try:
            return func()
        except RateLimitError:
            if attempt == max_retries - 1:
                raise
            wait_time = backoff_factor ** attempt
            print(f"Rate limit hit, waiting {wait_time} seconds...")
            time.sleep(wait_time)
        except APIError as e:
            print(f"API Error: {e}")
            if attempt == max_retries - 1:
                raise
            time.sleep(backoff_factor ** attempt)

# Usage
def safe_chat_completion(message):
    return robust_api_call(
        lambda: client.chat.completions.create(
            model="gpt-4",
            messages=[{"role": "user", "content": message}]
        )
    )
```

Token Management

```python
import tiktoken

def count_tokens(text, model="gpt-4"):
    """Count tokens in text"""
    encoding = tiktoken.encoding_for_model(model)
    return len(encoding.encode(text))

def truncate_text(text, max_tokens, model="gpt-4"):
    """Truncate text to fit within token limit"""
    encoding = tiktoken.encoding_for_model(model)
    tokens = encoding.encode(text)

    if len(tokens) <= max_tokens:
        return text

    truncated_tokens = tokens[:max_tokens]
    return encoding.decode(truncated_tokens)

def smart_chunking(text, chunk_size=1000, model="gpt-4"):
    """Split text into chunks based on token count"""
    encoding = tiktoken.encoding_for_model(model)
    tokens = encoding.encode(text)

    chunks = []
    for i in range(0, len(tokens), chunk_size):
        chunk_tokens = tokens[i:i + chunk_size]
        chunk_text = encoding.decode(chunk_tokens)
        chunks.append(chunk_text)

    return chunks
```

Cost Optimization

```
class CostTracker:
    def __init__(self):
        self.costs = {
            "gpt-4": {"input": 0.03, "output": 0.06},  # per 1K tokens
            "gpt-3.5-turbo": {"input": 0.001, "output": 0.002},
            "text-embedding-3-large": {"input": 0.00013, "output": 0}
        }
        self.total_cost = 0

    def calculate_cost(self, model, input_tokens, output_tokens):
        if model in self.costs:
            cost = (
                (input_tokens / 1000) * self.costs[model]["input"] +
                (output_tokens / 1000) * self.costs[model]["output"]
            )
            self.total_cost += cost
            return cost
        return 0

    def tracked_completion(self, **kwargs):
        response = client.chat.completions.create(**kwargs)

        usage = response.usage
        cost = self.calculate_cost(
            kwargs["model"],
            usage.prompt_tokens,
            usage.completion_tokens
        )

        print(f"Cost: ${cost:.4f} | Total: ${self.total_cost:.4f}")
        return response

# Usage
tracker = CostTracker()
response = tracker.tracked_completion(
    model="gpt-4",
    messages=[{"role": "user", "content": "Hello!"}]
)
```

Async Operations

```
import asyncio
from openai import AsyncOpenAI

async_client = AsyncOpenAI()

async def async_chat_completion(message):
    """Async chat completion"""
    response = await async_client.chat.completions.create(
        model="gpt-4",
        messages=[{"role": "user", "content": message}]
    )
    return response.choices[0].message.content

async def batch_completions(messages):
    """Process multiple completions concurrently"""
    tasks = [async_chat_completion(msg) for msg in messages]
    results = await asyncio.gather(*tasks)
    return results

# Usage
async def main():
    messages = [
        "What is Python?",
        "What is JavaScript?",
        "What is Rust?"
    ]

    results = await batch_completions(messages)
    for i, result in enumerate(results):
        print(f"Question {i+1}: {result[:100]}...")

# Run
# asyncio.run(main())
```

Production Configuration

```
class ProductionOpenAI:
    def __init__(self, api_key=None):
        self.client = OpenAI(
            api_key=api_key or os.getenv("OPENAI_API_KEY"),
            timeout=30,
            max_retries=3
        )
        self.default_params = {
            "temperature": 0.7,
            "max_tokens": 1000,
            "top_p": 0.9
        }

    def chat(self, messages, **kwargs):
        params = {**self.default_params, **kwargs}

        try:
            response = self.client.chat.completions.create(
                messages=messages,
                **params
            )
            return {
                "success": True,
                "content": response.choices[0].message.content,
                "usage": response.usage,
                "model": response.model
            }
        except Exception as e:
            return {
                "success": False,
                "error": str(e),
                "content": None
            }
```

Conclusion

The OpenAI API provides powerful capabilities for building AI-powered applications. This tutorial covered the essential patterns and best practices for:

- Chat completions and conversation management
- Function calling for tool integration
- Embeddings for semantic search
- Vision capabilities for image analysis
- Audio processing with Whisper and TTS
- Fine-tuning for specialized models
- Production-ready error handling and optimization

Next Steps:

1. Experiment with different models and parameters
2. Build a complete application using multiple API features
3. Implement proper monitoring and cost tracking
4. Explore advanced techniques like RAG and agent frameworks

Additional Resources:

- OpenAI API Documentation (https://platform.openai.com/docs)
- OpenAI Cookbook (https://github.com/openai/openai-cookbook)
- Best Practices Guide (https://platform.openai.com/docs/guides/production-best-practices)