```
import base64
import json
import logging
import os
from typing import Optional

import aiohttp
import requests
from dotenv import load_dotenv
from termcolor import colored
from loguru import logger
from swarm_models.base_multimodal_model import BaseMultiModalModel

# Load environment variables
load_dotenv()
openai_api_key = os.getenv("OPENAI_API_KEY")


gpt4_vision_system_prompt = """
You are an multi-modal autonomous agent. You are given a task and an image. You must generate a response to the task and image.

"""


class GPT4VisionAPI(BaseMultiModalModel):
```

"""

GPT-4 Vision API

This class is a wrapper for the OpenAI API. It is used to run the GPT-4 Vision model.

Parameters

----

openai_api_key : str

    The OpenAI API key. Defaults to the OPENAI_API_KEY environment variable.

max_tokens : int

    The maximum number of tokens to generate. Defaults to 300.

Methods

----

encode_image(img: str)

    Encode image to base64.

run(task: str, img: str)

    Run the model.

__call__(task: str, img: str)

    Run the model.

Examples:

>>> from swarm_models import GPT4VisionAPI

>>> llm = GPT4VisionAPI()

```
>>> task = "What is the color of the object?"
>>> img = "https://i.imgur.com/2M2ZGwC.jpeg"
>>> llm.run(task, img)



"""

def __init__(
    self,
    openai_api_key: str = openai_api_key,
    model_name: str = "gpt-4-vision-preview",
    logging_enabled: bool = False,
    max_workers: int = 10,
    max_tokens: str = 300,
    openai_proxy: str = "https://api.openai.com/v1/chat/completions",
    beautify: bool = False,
    streaming_enabled: Optional[bool] = False,
    meta_prompt: Optional[bool] = False,
    system_prompt: Optional[str] = gpt4_vision_system_prompt,
    *args,
    **kwargs,
):
    super(GPT4VisionAPI).__init__(*args, **kwargs)
    self.openai_api_key = openai_api_key
    self.logging_enabled = logging_enabled
    self.model_name = model_name
```

```
        self.max_workers = max_workers
        self.max_tokens = max_tokens
        self.openai_proxy = openai_proxy
        self.beautify = beautify
        self.streaming_enabled = streaming_enabled
        self.meta_prompt = meta_prompt
        self.system_prompt = system_prompt

        if self.logging_enabled:
            logging.basicConfig(level=logging.DEBUG)
        else:
            # Disable debug logs for requests and urllib3
            logging.getLogger("requests").setLevel(logging.WARNING)
            logging.getLogger("urllib3").setLevel(logging.WARNING)

        if self.meta_prompt:
            self.system_prompt = self.meta_prompt_init()

    def encode_image(self, img: str):
        """Encode image to base64."""
        if not os.path.exists(img):
            print(f"Image file not found: {img}")
            return None

        with open(img, "rb") as image_file:
            return base64.b64encode(image_file.read()).decode("utf-8")
```

```
def download_img_then_encode(self, img: str):
    """Download image from URL then encode image to base64 using requests"""
    if not os.path.exists(img):
        print(f"Image file not found: {img}")
        return None

    response = requests.get(img)
    return base64.b64encode(response.content).decode("utf-8")

# Function to handle vision tasks
def run(
    self,
    task: str = None,
    img: str = None,
    multi_imgs: list = None,
    return_json: bool = False,
    *args,
    **kwargs,
):
    """Run the model."""
    try:
        base64_image = self.encode_image(img)
        headers = {
            "Content-Type": "application/json",
            "Authorization": f"Bearer {self.openai_api_key}",
```

```
}
payload = {
    "model": self.model_name,
    "messages": [
        {
            "role": "system",
            "content": [self.system_prompt],
        },
        {
            "role": "user",
            "content": [
                {"type": "text", "text": task},
                {
                    "type": "image_url",
                    "image_url": {
                        "url": f"{base64_image}"
                    },
                },
            ],
        },
    ],
    "max_tokens": self.max_tokens,
    **kwargs,
}
response = requests.post(headers=headers, json=payload)
```

```
        # Get the response as a JSON object
        response_json = response.json()

        # Return the JSON object if return_json is True
        if return_json is True:
            print(response_json)
            return response_json
        else:
            return response_json

    except Exception as error:
        logger.error(
            f"Error with the request: {error}, make sure you"
            " double check input types and positions"
        )
        raise error

def video_prompt(self, frames):
    """
    SystemPrompt is a class that generates a prompt for the user to respond to.
    The prompt is generated based on the current state of the system.

    Parameters
    ----------
    frames : list
        A list of base64 frames
```

```
        Returns

        -------

        PROMPT : str

            The system prompt


        Examples

        --------


        >>> from swarm_models import GPT4VisionAPI

        >>> llm = GPT4VisionAPI()

        >>> video = "video.mp4"

        >>> base64_frames = llm.process_video(video)

        >>> prompt = llm.video_prompt(base64_frames)

        >>> print(prompt)


        """

        PROMPT = f"""

         These are frames from a video that I want to upload. Generate a compelling description that I can upload along with the video:


        {frames}

        """

        return PROMPT


    def stream_response(self, content: str):
```

```
    """Stream the response of the output

    Args:
        content (str): _description_
    """
    for chunk in content:
        print(chunk)

def __call__(
    self,
    task: Optional[str] = None,
    img: Optional[str] = None,
    *args,
    **kwargs,
):
    """Call the model

    Args:
        task (Optional[str], optional): _description_. Defaults to None.
        img (Optional[str], optional): _description_. Defaults to None.

    Raises:
        error: _description_
    """
    try:
        base64_image = self.encode_image(img)
```

```
headers = {
    "Content-Type": "application/json",
    "Authorization": f"Bearer {openai_api_key}",
}
payload = {
    "model": self.model_name,
    "messages": [
        {
            "role": "system",
            "content": [self.system_prompt],
        },
        {
            "role": "user",
            "content": [
                {"type": "text", "text": task},
                {
                    "type": "image_url",
                    "image_url": {
                        "url": f"{base64_image}"
                    },
                },
            ],
        },
    ],
    "max_tokens": self.max_tokens,
}
```

```
        response = requests.post(
            self.openai_proxy,
            headers=headers,
            json=payload,
        )

        out = response.json()
        content = out["choices"][0]["message"]["content"]

        if self.streaming_enabled:
            content = self.stream_response(content)

        if self.beautify:
            content = colored(content, "cyan")
            print(content)
        else:
            print(content)

    except Exception as error:
        print(f"Error with the request: {error}")
        raise error

async def arun(
    self,
    task: Optional[str] = None,
    img: Optional[str] = None,
```

```
):
    """
    Asynchronously run the model

    Overview:
    ---------
    This method is used to asynchronously run the model. It is used to run the model
    on a single task and image.

    Parameters:
    ----------
    task : str
        The task to run the model on.
    img : str
        The image to run the task on

    """
    try:
        base64_image = self.encode_image(img)
        headers = {
            "Content-Type": "application/json",
            "Authorization": f"Bearer {openai_api_key}",
        }
        payload = {
            "model": "gpt-4-vision-preview",
            "messages": [
```

```
            {
                "role": "user",
                "content": [
                    {"type": "text", "text": task},
                    {
                        "type": "image_url",
                        "image_url": {
                            "url": f"{base64_image}"
                        },
                    },
                ],
            }
        ],
        "max_tokens": self.max_tokens,
    }
    async with aiohttp.ClientSession() as session:
        async with session.post(
            self.openai_proxy,
            headers=headers,
            data=json.dumps(payload),
        ) as response:
            out = await response.json()
            content = out["choices"][0]["message"]["content"]
            print(content)
except Exception as error:
    print(f"Error with the request {error}")
```

```
        raise error


def health_check(self):
    """Health check for the GPT4Vision model"""
    try:
        response = requests.get(
            "https://api.openai.com/v1/engines"
        )
        return response.status_code == 200
    except requests.RequestException as error:
        print(f"Health check failed: {error}")
        return False


def print_dashboard(self):
    dashboard = print(
        colored(
            f"""
        GPT4Vision Dashboard
        -------------------
        Model: {self.model_name}
        Max Workers: {self.max_workers}
        OpenAIProxy: {self.openai_proxy}
        """,
            "green",
        )
    )
```

```
    return dashboard


# def meta_prompt_init(self):
#     """Meta Prompt


#     Returns:
#         _type_: _description_
#     """
#     META_PROMPT = """
#     For any labels or markings on an image that you reference in your response, please
#     enclose them in square brackets ([]) and list them explicitly. Do not use ranges; for
#     example, instead of '1 - 4', list as '[1], [2], [3], [4]'. These labels could be
#     numbers or letters and typically correspond to specific segments or parts of the image.
#     """
#     return META_PROMPT
```