```
{
 "cells": [
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {
    "id": "cs5RHepmhkEh"
   },
   "outputs": [],
   "source": [
    "!pip3 install -U swarms python-dotenv"
   ]
  },
  {
   "cell_type": "markdown",
   "metadata": {
    "id": "-d9k3egzgp2_"
   },
   "source": [
    "Copied from the repo, example.py\n",
    "Enter your OpenAI API key here."
   ]
  },
  {
   "cell_type": "markdown",
   "metadata": {},
 "source": [
  "A basic example of how to use the OpenAI API to generate text."
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "import os\n",
  "\n",
  "from dotenv import load_dotenv\n",
  "\n",
  "# Import the OpenAIChat model and the Agent struct\n",
  "from swarms import Agent, OpenAIChat\n",
  "\n",
  "# Load the environment variables\n",
  "load_dotenv()\n",
  "\n",
  "# Get the API key from the environment\n",
  "api_key = os.environ.get(\"OPENAI_API_KEY\")\n",
  "\n",
  "# Initialize the language model\n",
  "llm = OpenAIChat(\n",
  "    temperature=0.5, openai_api_key=api_key, max_tokens=4000\n",
   ")\n",
   "\n",
   "\n",
   "## Initialize the workflow\n",
   "agent = Agent(llm=llm, max_loops=1, autosave=True, dashboard=True)\n",
   "\n",
   "# Run the workflow on a task\n",
   "agent.run(\"Generate a 10,000 word blog on health and wellness.\")"
  ]
 },
 {
  "cell_type": "markdown",
  "metadata": {
   "id": "6VtgQ0F4BNc-"
  },
  "source": [
   "Look at the log, which may be empty."
  ]
 },
 {
  "cell_type": "code",
  "execution_count": null,
  "metadata": {
   "id": "RqL5LL3xBLWR"
  },
  "outputs": [],
   "source": [
    "!cat errors.txt"
   ]
  },
  {
   "cell_type": "markdown",
   "metadata": {},
   "source": [
    "**Agent with Long Term Memory**\n",
    "\n",
    "```Agent``` equipped with quasi-infinite long term memory. Great for long document understanding, analysis, and retrieval."
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {},
   "outputs": [],
   "source": [
    "from swarms import Agent, OpenAIChat\n",
    "from swarms_memory import ChromaDB\n",
    "\n",
    "# Making an instance of the ChromaDB class\n",
    "memory = ChromaDB(\n",
    "    metric=\"cosine\",\n",
```

```
"    n_results=3,\n",
"    output_dir=\"results\",\n",
"    docs_folder=\"docs\",\n",
")\n",
"\n",
"# Initializing the agent with the OpenAI instance and other parameters\n",
"agent = Agent(\n",
"    agent_name=\"Covid-19-Chat\",\n",
"    agent_description=(\n",
"        \"This agent provides information about COVID-19 symptoms.\"\n",
"    ),\n",
"    llm=OpenAIChat(),\n",
"    max_loops=\"auto\",\n",
"    autosave=True,\n",
"    verbose=True,\n",
"    long_term_memory=memory,\n",
"    stopping_condition=\"finish\",\n",
")\n",
"\n",
"# Defining the task and image path\n",
"task = (\"What are the symptoms of COVID-19?\",)\n",
"\n",
"# Running the agent with the specified task and image\n",
"out = agent.run(task)\n",
"print(out)"
]
```

```
 },
 {
  "cell_type": "markdown",
  "metadata": {},
  "source": [
   "**```Agent``` with Long Term Memory ++ Tools!**\n",
    "An LLM equipped with long term memory and tools, a full stack agent capable of automating all
and any digital tasks given a good prompt."
  ]
 },
 {
  "cell_type": "code",
  "execution_count": null,
  "metadata": {},
  "outputs": [],
  "source": [
   "# !pip install swarms-memory\n",
   "from swarms import Agent, OpenAIChat, tool\n",
   "from swarms_memory import ChromaDB\n",
   "\n",
   "# Making an instance of the ChromaDB class\n",
   "memory = ChromaDB(\n",
   "    metric=\"cosine\",\n",
   "    n_results=3,\n",
   "    output_dir=\"results\",\n",
   "    docs_folder=\"docs\",\n",
```

```
")\n",
"\n",
"# Initialize a tool\n",
"@tool\n",
"def search_api(query: str):\n",
"    # Add your logic here\n",
"    return query\n",
"\n",
"# Initializing the agent with the OpenAI instance and other parameters\n",
"agent = Agent(\n",
"    agent_name=\"Covid-19-Chat\",\n",
"    agent_description=(\n",
"        \"This agent provides information about COVID-19 symptoms.\"\n",
"    ),\n",
"    llm=OpenAIChat(),\n",
"    max_loops=\"auto\",\n",
"    autosave=True,\n",
"    verbose=True,\n",
"    long_term_memory=memory,\n",
"    stopping_condition=\"finish\",\n",
"    tools=[search_api],\n",
")\n",
"\n",
"# Defining the task and image path\n",
"task = (\"What are the symptoms of COVID-19?\",)\n",
"\n",
    "# Running the agent with the specified task and image\n",
    "out = agent.run(task)\n",
    "print(out)\n"
   ]
  },
  {
   "cell_type": "markdown",
   "metadata": {},
   "source": [
    "**Simple Conversational Agent**\n",
    "A Plug in and play conversational agent with GPT4, Mixytral, or any of our models\n",
    "\n",
    "     - Reliable conversational structure to hold messages together with dynamic handling for long
context conversations and interactions with auto chunking\n",
    "\n",
    "    - Reliable, this simple system will always provide responses you want.\n"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {},
   "outputs": [],
   "source": [
    "from swarms import Agent
from swarm_models import Anthropic\n",
  "\n",
  "\n",
  "## Initialize the workflow\n",
  "agent = Agent(\n",
  "    agent_name=\"Transcript Generator\",\n",
  "    agent_description=(\n",
  "        \"Generate a transcript for a youtube video on what swarms\"\n",
  "        \" are!\"\n",
  "    ),\n",
  "    llm=Anthropic(),\n",
  "    max_loops=3,\n",
  "    autosave=True,\n",
  "    dashboard=False,\n",
  "    streaming_on=True,\n",
  "    verbose=True,\n",
  "    stopping_token=\"<DONE>\",\n",
  "    interactive=True, # Set to True\n",
  ")\n",
  "\n",
  "# Run the workflow on a task\n",
  "agent(\"Generate a transcript for a youtube video on what swarms are!\")"
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
```

"source": [
 "**Devin**\n",
 "\n",
 "Implementation of Devil in less than 90 lines of code with several tools: terminal, browser, and edit files!"
]
},
{
"cell_type": "code",
"execution_count": null,
"metadata": {},
"outputs": [],
"source": [
 "from swarms import Agent
from swarm_models import Anthropic, tool\n",
 "import subprocess\n",
 "\n",
 "# Model\n",
 "llm = Anthropic(\n",
 "   temperature=0.1,\n",
 ")\n",
 "\n",
 "# Tools\n",
 "@tool\n",
 "def terminal(\n",
 "   code: str,\n",

```
"):\n",
"    \"\"\"\n",
"    Run code in the terminal.\n",
"\n",
"    Args:\n",
"        code (str): The code to run in the terminal.\n",
"\n",
"    Returns:\n",
"        str: The output of the code.\n",
"    \"\"\"\n",
"    out = subprocess.run(\n",
"        code, shell=True, capture_output=True, text=True\n",
"    ).stdout\n",
"    return str(out)\n",
"\n",
"\n",
"@tool\n",
"def browser(query: str):\n",
"    \"\"\"\n",
"    Search the query in the browser with the `browser` tool.\n",
"\n",
"    Args:\n",
"        query (str): The query to search in the browser.\n",
"\n",
"    Returns:\n",
"        str: The search results.\n",
"    \"\"\"\n",
"    import webbrowser\n",
"\n",
"    url = f\"https://www.google.com/search?q={query}\"\n",
"    webbrowser.open(url)\n",
"    return f\"Searching for {query} in the browser.\"\n",
"\n",
"@tool\n",
"def create_file(file_path: str, content: str):\n",
"    \"\"\"\n",
"    Create a file using the file editor tool.\n",
"\n",
"    Args:\n",
"        file_path (str): The path to the file.\n",
"        content (str): The content to write to the file.\n",
"\n",
"    Returns:\n",
"        str: The result of the file creation operation.\n",
"    \"\"\"\n",
"    with open(file_path, \"w\") as file:\n",
"        file.write(content)\n",
"    return f\"File {file_path} created successfully.\"\n",
"\n",
"@tool\n",
"def file_editor(file_path: str, mode: str, content: str):\n",
"    \"\"\"\n",
"    Edit a file using the file editor tool.\n",
"\n",
"    Args:\n",
"        file_path (str): The path to the file.\n",
"        mode (str): The mode to open the file in.\n",
"        content (str): The content to write to the file.\n",
"\n",
"    Returns:\n",
"        str: The result of the file editing operation.\n",
"    \"\"\"\n",
"    with open(file_path, mode) as file:\n",
"        file.write(content)\n",
"    return f\"File {file_path} edited successfully.\"\n",
"\n",
"\n",
"# Agent\n",
"agent = Agent(\n",
"    agent_name=\"Devin\",\n",
"    system_prompt=(\n",
"        \"Autonomous agent that can interact with humans and other\"\n",
"        \" agents. Be Helpful and Kind. Use the tools provided to\"\n",
"        \" assist the user. Return all code in markdown format.\"\n",
"    ),\n",
"    llm=llm,\n",
"    max_loops=\"auto\",\n",
"    autosave=True,\n",
    "    dashboard=False,\n",
    "    streaming_on=True,\n",
    "    verbose=True,\n",
    "    stopping_token=\"<DONE>\",\n",
    "    interactive=True,\n",
    "    tools=[terminal, browser, file_editor, create_file],\n",
    "    code_interpreter=True,\n",
    "    # streaming=True,\n",
    ")\n",
    "\n",
    "# Run the agent\n",
    "out = agent(\"Create a new file for a plan to take over the world.\")\n",
    "print(out)"
   ]
  },
  {
   "cell_type": "markdown",
   "metadata": {},
   "source": [
    "**Agentwith Pydantic BaseModel as Output Type**\n",
    "\n",
    "The following is an example of an agent that intakes a pydantic basemodel and outputs it at the same time:"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {},
   "outputs": [],
   "source": [
    "from pydantic import BaseModel, Field\n",
    "from swarms import Agent
from swarm_models import Anthropic\n",
    "\n",
    "\n",
    "# Initialize the schema for the person's information\n",
    "class Schema(BaseModel):\n",
    "    name: str = Field(..., title=\"Name of the person\")\n",
    "    agent: int = Field(..., title=\"Age of the person\")\n",
    "    is_student: bool = Field(..., title=\"Whether the person is a student\")\n",
    "    courses: list[str] = Field(\n",
    "        ..., title=\"List of courses the person is taking\"\n",
    "    )\n",
    "\n",
    "\n",
    "# Convert the schema to a JSON string\n",
    "tool_schema = Schema(\n",
    "    name=\"Tool Name\",\n",
    "    agent=1,\n",
    "    is_student=True,\n",
    "    courses=[\"Course1\", \"Course2\"],\n",
")\n",
"\n",
"# Define the task to generate a person's information\n",
"task = \"Generate a person's information based on the following schema:\"\n",
"\n",
"# Initialize the agent\n",
"agent = Agent(\n",
"    agent_name=\"Person Information Generator\",\n",
"    system_prompt=(\n",
"        \"Generate a person's information based on the following schema:\"\n",
"    ),\n",
"    # Set the tool schema to the JSON string -- this is the key difference\n",
"    tool_schema=tool_schema,\n",
"    llm=Anthropic(),\n",
"    max_loops=3,\n",
"    autosave=True,\n",
"    dashboard=False,\n",
"    streaming_on=True,\n",
"    verbose=True,\n",
"    interactive=True,\n",
"    # Set the output type to the tool schema which is a BaseModel\n",
"    output_type=tool_schema,  # or dict, or str\n",
"    metadata_output_type=\"json\",\n",
"    # List of schemas that the agent can handle\n",
"    list_base_models=[tool_schema],\n",
"    function_calling_format_type=\"OpenAI\",\n",
```

"   function_calling_type=\"json\", # or soon yaml\n",

")\n",

"\n",

"# Run the agent to generate the person's information\n",

"generated_data = agent.run(task)\n",

"\n",

"# Print the generated data\n",

"print(f\"Generated data: {generated_data}\")\n",

"\n"

]

},

{

"cell_type": "markdown",

"metadata": {},

"source": [

"**```ToolAgent```**\n",

"\n",

"ToolAgent is an agent that can use tools through JSON function calling. It intakes any open source model from huggingface and is extremely modular and plug in and play. We need help adding general support to all models soon."

]

},

{

"cell_type": "code",

"execution_count": null,

"metadata": {},

```
"outputs": [],
"source": [
 "from pydantic import BaseModel, Field\n",
 "from transformers import AutoModelForCausalLM, AutoTokenizer\n",
 "\n",
 "from swarms import ToolAgent\n",
 "from swarms.utils.json_utils import base_model_to_json\n",
 "\n",
 "# Load the pre-trained model and tokenizer\n",
 "model = AutoModelForCausalLM.from_pretrained(\n",
 "    \"databricks/dolly-v2-12b\",\n",
 "    load_in_4bit=True,\n",
 "    device_map=\"auto\",\n",
 ")\n",
 "tokenizer = AutoTokenizer.from_pretrained(\"databricks/dolly-v2-12b\")\n",
 "\n",
 "\n",
 "# Initialize the schema for the person's information\n",
 "class Schema(BaseModel):\n",
 "    name: str = Field(..., title=\"Name of the person\")\n",
 "    agent: int = Field(..., title=\"Age of the person\")\n",
 "    is_student: bool = Field(\n",
 "        ..., title=\"Whether the person is a student\"\n",
 "    )\n",
 "    courses: list[str] = Field(\n",
 "        ..., title=\"List of courses the person is taking\"\n",
"    )\n",
"\n",
"\n",
"# Convert the schema to a JSON string\n",
"tool_schema = base_model_to_json(Schema)\n",
"\n",
"# Define the task to generate a person's information\n",
"task = (\n",
"    \"Generate a person's information based on the following schema:\"\n",
")\n",
"\n",
"# Create an instance of the ToolAgent class\n",
"agent = ToolAgent(\n",
"    name=\"dolly-function-agent\",\n",
"    description=\"Ana gent to create a child data\",\n",
"    model=model,\n",
"    tokenizer=tokenizer,\n",
"    json_schema=tool_schema,\n",
")\n",
"\n",
"# Run the agent to generate the person's information\n",
"generated_data = agent.run(task)\n",
"\n",
"# Print the generated data\n",
"print(f\"Generated data: {generated_data}\")\n"
]
```

```
  },
  {
   "cell_type": "markdown",
   "metadata": {},
   "source": [
    "**Worker**\n",
    "\n",
    "The Worker is a simple all-in-one agent equipped with an LLM, tools, and RAG for low level tasks.\n",
    "\n",
    " Plug in and Play LLM. Utilize any LLM from anywhere and any framework\n",
    "\n",
    " Reliable RAG: Utilizes FAISS for efficient RAG but it's modular so you can use any DB.\n",
    "\n",
    " Multi-Step Parallel Function Calling: Use any tool"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {},
   "outputs": [],
   "source": [
    "# Importing necessary modules\n",
    "import os\n",
    "\n",
"from dotenv import load_dotenv\n",

"\n",

"from swarm_models import OpenAIChat, Worker, tool\n",

"\n",

"# Loading environment variables from .env file\n",

"load_dotenv()\n",

"\n",

"# Retrieving the OpenAI API key from environment variables\n",

"api_key = os.getenv(\"OPENAI_API_KEY\")\n",

"\n",

"\n",

"# Create a tool\n",

"@tool\n",

"def search_api(query: str):\n",

"    pass\n",

"\n",

"\n",

"# Creating a Worker instance\n",

"worker = Worker(\n",

"    name=\"My Worker\",\n",

"    role=\"Worker\",\n",

"    human_in_the_loop=False,\n",

"    tools=[search_api],\n",

"    temperature=0.5,\n",

"    llm=OpenAIChat(openai_api_key=api_key),\n",

")\n",
    "\n",
    "# Running the worker with a prompt\n",
    "out = worker.run(\"Hello, how are you? Create an image of how your are doing!\")\n",
    "\n",
    "# Printing the output\n",
    "print(out)"
   ]
  },
  {
   "cell_type": "markdown",
   "metadata": {},
   "source": [
    "**```SequentialWorkflow```**\n",
    "\n",
     "Sequential Workflow enables you to sequentially execute tasks with Agent and then pass the output into the next agent and onwards until you have specified your max loops. ```SequentialWorkflow``` is wonderful for real-world business tasks like sending emails, summarizing documents, and analyzing data.\n",
    "\n",
    " Save and Restore Workflow states!\n",
    "\n",
    " Multi-Modal Support for Visual Chaining\n",
    "\n",
    " Utilizes Agent class"
   ]
  },
```

```
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "import os\n",
  "\n",
  "from dotenv import load_dotenv\n",
  "\n",
  "from swarms import Agent, OpenAIChat, SequentialWorkflow\n",
  "\n",
  "load_dotenv()\n",
  "\n",
  "# Load the environment variables\n",
  "api_key = os.getenv(\"OPENAI_API_KEY\")\n",
  "\n",
  "\n",
  "# Initialize the language agent\n",
  "llm = OpenAIChat(\n",
  "    temperature=0.5, model_name=\"gpt-4\", openai_api_key=api_key, max_tokens=4000\n",
  ")\n",
  "\n",
  "\n",
  "# Initialize the agent with the language agent\n",
  "agent1 = Agent(llm=llm, max_loops=1)\n",
"\n",
"# Create another agent for a different task\n",
"agent2 = Agent(llm=llm, max_loops=1)\n",
"\n",
"# Create another agent for a different task\n",
"agent3 = Agent(llm=llm, max_loops=1)\n",
"\n",
"# Create the workflow\n",
"workflow = SequentialWorkflow(max_loops=1)\n",
"\n",
"# Add tasks to the workflow\n",
"workflow.add(\n",
"    agent1,\n",
"    \"Generate a 10,000 word blog on health and wellness.\",\n",
")\n",
"\n",
"# Suppose the next task takes the output of the first task as input\n",
"workflow.add(\n",
"    agent2,\n",
"    \"Summarize the generated blog\",\n",
")\n",
"\n",
"# Run the workflow\n",
"workflow.run()\n",
"\n",
"# Output the results\n",
   "for task in workflow.tasks:\n",
   "    print(f\"Task: {task.description}, Result: {task.result}\")"
  ]
 },
 {
  "cell_type": "markdown",
  "metadata": {},
  "source": [
   "**```ConcurrentWorkflow```**\n",
   "\n",
   "```ConcurrentWorkflow``` runs all the tasks all at the same time with the inputs you give it!"
  ]
 },
 {
  "cell_type": "code",
  "execution_count": null,
  "metadata": {},
  "outputs": [],
  "source": [
   "import os\n",
   "\n",
   "from dotenv import load_dotenv\n",
   "\n",
   "from swarms import Agent, ConcurrentWorkflow, OpenAIChat, Task\n",
   "\n",
   "# Load environment variables from .env file\n",
```

```
   "load_dotenv()\n",
   "\n",
   "# Load environment variables\n",
   "llm = OpenAIChat(openai_api_key=os.getenv(\"OPENAI_API_KEY\"))\n",
   "agent = Agent(llm=llm, max_loops=1)\n",
   "\n",
   "# Create a workflow\n",
   "workflow = ConcurrentWorkflow(max_workers=5)\n",
   "\n",
   "# Create tasks\n",
   "task1 = Task(agent, \"What's the weather in miami\")\n",
   "task2 = Task(agent, \"What's the weather in new york\")\n",
   "task3 = Task(agent, \"What's the weather in london\")\n",
   "\n",
   "# Add tasks to the workflow\n",
   "workflow.add(tasks=[task1, task2, task3])\n",
   "\n",
   "# Run the workflow\n",
   "workflow.run()"
  ]
 },
 {
  "cell_type": "markdown",
  "metadata": {},
  "source": [
   "**```RecursiveWorkflow```**\n",
```

```
    "\n",
     "```RecursiveWorkflow``` will keep executing the tasks until a specific token like is located inside
the text!"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {},
   "outputs": [],
   "source": [
    "import os\n",
    "\n",
    "from dotenv import load_dotenv\n",
    "\n",
    "from swarms import Agent, OpenAIChat, RecursiveWorkflow, Task\n",
    "\n",
    "# Load environment variables from .env file\n",
    "load_dotenv()\n",
    "\n",
    "# Load environment variables\n",
    "llm = OpenAIChat(openai_api_key=os.getenv(\"OPENAI_API_KEY\"))\n",
    "agent = Agent(llm=llm, max_loops=1)\n",
    "\n",
    "# Create a workflow\n",
    "workflow = RecursiveWorkflow(stop_token=\"<DONE>\")\n",
```

```
    "\n",
    "# Create tasks\n",
    "task1 = Task(agent, \"What's the weather in miami\")\n",
    "task2 = Task(agent, \"What's the weather in new york\")\n",
    "task3 = Task(agent, \"What's the weather in london\")\n",
    "\n",
    "# Add tasks to the workflow\n",
    "workflow.add(task1)\n",
    "workflow.add(task2)\n",
    "workflow.add(task3)\n",
    "\n",
    "# Run the workflow\n",
    "workflow.run()"
   ]
  },
  {
   "cell_type": "markdown",
   "metadata": {},
   "source": [
    "**```ModelParallelizer```**\n",
    "\n",
    "The ```ModelParallelizer``` allows you to run multiple models concurrently, comparing their outputs. This feature enables you to easily compare the performance and results of different models, helping you make informed decisions about which model to use for your specific task.\n",
    "\n",
    "Plug-and-Play Integration: The structure provides a seamless integration with various models,
```

including OpenAIChat, Anthropic, Mixtral, and Gemini. You can easily plug in any of these models and start using them without the need for extensive modifications or setup."
```
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {},
   "outputs": [],
   "source": [
    "import os\n",
    "\n",
    "from dotenv import load_dotenv\n",
    "\n",
    "from swarms import Anthropic, Gemini, Mixtral, ModelParallelizer, OpenAIChat\n",
    "\n",
    "load_dotenv()\n",
    "\n",
    "# API Keys\n",
    "anthropic_api_key = os.getenv(\"ANTHROPIC_API_KEY\")\n",
    "openai_api_key = os.getenv(\"OPENAI_API_KEY\")\n",
    "gemini_api_key = os.getenv(\"GEMINI_API_KEY\")\n",
    "\n",
    "# Initialize the models\n",
    "llm = OpenAIChat(openai_api_key=openai_api_key)\n",
    "anthropic = Anthropic(anthropic_api_key=anthropic_api_key)\n",
```

"mixtral = Mixtral()\n",

"gemini = Gemini(gemini_api_key=gemini_api_key)\n",

"\n",

"# Initialize the parallelizer\n",

"llms = [llm, anthropic, mixtral, gemini]\n",

"parallelizer = ModelParallelizer(llms)\n",

"\n",

"# Set the task\n",

"task = \"Generate a 10,000 word blog on health and wellness.\"\n",

"\n",

"# Run the task\n",

"out = parallelizer.run(task)\n",

"\n",

"# Print the responses 1 by 1\n",

"for i in range(len(out)):\n",

"   print(f\"Response from LLM {i}: {out[i]}\")"

]

},

{

"cell_type": "markdown",

"metadata": {},

"source": [

"**```SwarmNetwork```**\n",

"\n",

"```SwarmNetwork``` provides the infrasturcture for building extremely dense and complex multi-agent applications that span across various types of agents.\n",

"\n",
" Efficient Task Management: ```SwarmNetwork```'s intelligent agent pool and task queue management system ensures tasks are distributed evenly across agents. This leads to efficient use of resources and faster task completion.\n",
"\n",
" Scalability: ```SwarmNetwork``` can dynamically scale the number of agents based on the number of pending tasks. This means it can handle an increase in workload by adding more agents, and conserve resources when the workload is low by reducing the number of agents.\n",
"\n",
" Versatile Deployment Options: With ```SwarmNetwork```, each agent can be run on its own thread, process, container, machine, or even cluster. This provides a high degree of flexibility and allows for deployment that best suits the user's needs and infrastructure."
]
},
{
"cell_type": "code",
"execution_count": null,
"metadata": {},
"outputs": [],
"source": [
"import os\n",
"\n",
"from dotenv import load_dotenv\n",
"\n",
"# Import the OpenAIChat model and the Agent struct\n",
"from swarms import Agent, OpenAIChat, SwarmNetwork\n",

```
"\n",
"# Load the environment variables\n",
"load_dotenv()\n",
"\n",
"# Get the API key from the environment\n",
"api_key = os.environ.get(\"OPENAI_API_KEY\")\n",
"\n",
"# Initialize the language model\n",
"llm = OpenAIChat(\n",
"    temperature=0.5,\n",
"    openai_api_key=api_key,\n",
")\n",
"\n",
"## Initialize the workflow\n",
"agent = Agent(llm=llm, max_loops=1, agent_name=\"Social Media Manager\")\n",
"agent2 = Agent(llm=llm, max_loops=1, agent_name=\" Product Manager\")\n",
"agent3 = Agent(llm=llm, max_loops=1, agent_name=\"SEO Manager\")\n",
"\n",
"\n",
"# Load the swarmnet with the agents\n",
"swarmnet = SwarmNetwork(\n",
"    agents=[agent, agent2, agent3],\n",
")\n",
"\n",
"# List the agents in the swarm network\n",
"out = swarmnet.list_agents()\n",
    "print(out)\n",
    "\n",
    "# Run the workflow on a task\n",
    "out = swarmnet.run_single_agent(\n",
    "    agent2.id, \"Generate a 10,000 word blog on health and wellness.\"\n",
    ")\n",
    "print(out)\n",
    "\n",
    "\n",
    "# Run all the agents in the swarm network on a task\n",
    "out = swarmnet.run_many_agents(\"Generate a 10,000 word blog on health and wellness.\")\n",
    "print(out)"
   ]
  },
  {
   "cell_type": "markdown",
   "metadata": {},
   "source": [
    "\n",
    "**```Task```**\n",
    "\n",
    "```Task``` is a simple structure for task execution with the ```Agent```. Imagine zapier for LLM-based workflow automation\n",
    "\n",
    " ```Task``` is a structure for task execution with the ```Agent```.\n",
    "\n",
```

```
      " ```Tasks``` can have descriptions, scheduling, triggers, actions, conditions, dependencies, priority, and a history.\n",
    "\n",
    " The ```Task``` structure allows for efficient workflow automation with LLM-based agents."
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {},
   "outputs": [],
   "source": [
    "import os\n",
    "\n",
    "from dotenv import load_dotenv\n",
    "\n",
    "from swarms.structs import Agent, OpenAIChat, Task\n",
    "\n",
    "# Load the environment variables\n",
    "load_dotenv()\n",
    "\n",
    "\n",
    "# Define a function to be used as the action\n",
    "def my_action():\n",
    "    print(\"Action executed\")\n",
    "\n",
```

```
"\n",
"# Define a function to be used as the condition\n",
"def my_condition():\n",
"    print(\"Condition checked\")\n",
"    return True\n",
"\n",
"\n",
"# Create an agent\n",
"agent = Agent(\n",
"    llm=OpenAIChat(openai_api_key=os.environ[\"OPENAI_API_KEY\"]),\n",
"    max_loops=1,\n",
"    dashboard=False,\n",
")\n",
"\n",
"# Create a task\n",
"task = Task(\n",
"    description=(\n",
"        \"Generate a report on the top 3 biggest expenses for small\"\n",
"        \" businesses and how businesses can save 20%\"\n",
"    ),\n",
"    agent=agent,\n",
")\n",
"\n",
"# Set the action and condition\n",
"task.set_action(my_action)\n",
"task.set_condition(my_condition)\n",
    "\n",
    "# Execute the task\n",
    "print(\"Executing task...\")\n",
    "task.run()\n",
    "\n",
    "# Check if the task is completed\n",
    "if task.is_completed():\n",
    "    print(\"Task completed\")\n",
    "else:\n",
    "    print(\"Task not completed\")\n",
    "\n",
    "# Output the result of the task\n",
    "print(f\"Task result: {task.result}\")"
   ]
  },
  {
   "cell_type": "markdown",
   "metadata": {},
   "source": [
    "**```BlocksList```**\n",
    "\n",
    "    Modularity and Flexibility: ```BlocksList``` allows users to create custom swarms by adding or removing different classes or functions as blocks. This means users can easily tailor the functionality of their swarm to suit their specific needs.\n",
    "\n",
    "        Ease of Management: With methods to add, remove, update, and retrieve blocks,
```

```BlocksList``` provides a straightforward way to manage the components of a swarm. This makes it easier to maintain and update the swarm over time.\n",
    "\n",
     "   Enhanced Searchability: ```BlocksList``` offers methods to get blocks by various attributes such as name, type, ID, and parent-related properties. This makes it easier for users to find and work with specific blocks in a large and complex swarm.\n"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {},
   "outputs": [],
   "source": [
    "import os\n",
    "\n",
    "from dotenv import load_dotenv\n",
    "from transformers import AutoModelForCausalLM, AutoTokenizer\n",
    "from pydantic import BaseModel\n",
    "from swarms import BlocksList, Gemini, GPT4VisionAPI, Mixtral, OpenAI, ToolAgent\n",
    "\n",
    "# Load the environment variables\n",
    "load_dotenv()\n",
    "\n",
    "# Get the environment variables\n",
    "openai_api_key = os.getenv(\"OPENAI_API_KEY\")\n",

```
"gemini_api_key = os.getenv(\"GEMINI_API_KEY\")\n",

"\n",

"# Tool Agent\n",

"model = AutoModelForCausalLM.from_pretrained(\"databricks/dolly-v2-12b\")\n",

"tokenizer = AutoTokenizer.from_pretrained(\"databricks/dolly-v2-12b\")\n",

"\n",

"# Initialize the schema for the person's information\n",

"class Schema(BaseModel):\n",

"    name: str = Field(..., title=\"Name of the person\")\n",

"    agent: int = Field(..., title=\"Age of the person\")\n",

"    is_student: bool = Field(\n",

"        ..., title=\"Whether the person is a student\"\n",

"    )\n",

"    courses: list[str] = Field(\n",

"        ..., title=\"List of courses the person is taking\"\n",

"    )\n",

"\n",

"# Convert the schema to a JSON string\n",

"json_schema = base_model_to_json(Schema)\n",

"\n",

"\n",

"toolagent = ToolAgent(model=model, tokenizer=tokenizer, json_schema=json_schema)\n",

"\n",

"# Blocks List which enables you to build custom swarms by adding classes or functions\n",

"swarm = BlocksList(\n",

"    \"SocialMediaSwarm\",\n",
"    \"A swarm of social media agents\",\n",
"    [\n",
"        OpenAI(openai_api_key=openai_api_key),\n",
"        Mixtral(),\n",
"        GPT4VisionAPI(openai_api_key=openai_api_key),\n",
"        Gemini(gemini_api_key=gemini_api_key),\n",
"    ],\n",
")\n",
"\n",
"\n",
"# Add the new block to the swarm\n",
"swarm.add(toolagent)\n",
"\n",
"# Remove a block from the swarm\n",
"swarm.remove(toolagent)\n",
"\n",
"# Update a block in the swarm\n",
"swarm.update(toolagent)\n",
"\n",
"# Get a block at a specific index\n",
"block_at_index = swarm.get(0)\n",
"\n",
"# Get all blocks in the swarm\n",
"all_blocks = swarm.get_all()\n",
"\n",
"# Get blocks by name\n",
"openai_blocks = swarm.get_by_name(\"OpenAI\")\n",

"\n",

"# Get blocks by type\n",

"gpt4_blocks = swarm.get_by_type(\"GPT4VisionAPI\")\n",

"\n",

"# Get blocks by ID\n",

"block_by_id = swarm.get_by_id(toolagent.id)\n",

"\n",

"# Get blocks by parent\n",

"blocks_by_parent = swarm.get_by_parent(swarm)\n",

"\n",

"# Get blocks by parent ID\n",

"blocks_by_parent_id = swarm.get_by_parent_id(swarm.id)\n",

"\n",

"# Get blocks by parent name\n",

"blocks_by_parent_name = swarm.get_by_parent_name(swarm.name)\n",

"\n",

"# Get blocks by parent type\n",

"blocks_by_parent_type = swarm.get_by_parent_type(type(swarm).__name__)\n",

"\n",

"# Get blocks by parent description\n",

"blocks_by_parent_description = swarm.get_by_parent_description(swarm.description)\n",

"\n",

"# Run the block in the swarm\n",

"inference = swarm.run_block(toolagent, \"Hello World\")\n",

"print(inference)"
```

```
  ]
 },
 {
  "cell_type": "markdown",
  "metadata": {},
  "source": [
   "**Majority Voting**\n",
   "\n",
   "Multiple-agents will evaluate an idea based off of an parsing or evaluation function. From papers
like \"More agents is all you need\""
  ]
 },
 {
  "cell_type": "code",
  "execution_count": null,
  "metadata": {},
  "outputs": [],
  "source": [
   "from swarms import Agent, MajorityVoting, ChromaDB, Anthropic\n",
   "\n",
   "# Initialize the llm\n",
   "llm = Anthropic()\n",
   "\n",
   "# Agents\n",
   "agent1 = Agent(\n",
   "    llm = llm,\n",
"      system_prompt=\"You are the leader of the Progressive Party. What is your stance on healthcare?\",\n",
"    agent_name=\"Progressive Leader\",\n",
"    agent_description=\"Leader of the Progressive Party\",\n",
"    long_term_memory=ChromaDB(),\n",
"    max_steps=1,\n",
")\n",
"\n",
"agent2 = Agent(\n",
"    llm=llm,\n",
"    agent_name=\"Conservative Leader\",\n",
"    agent_description=\"Leader of the Conservative Party\",\n",
"    long_term_memory=ChromaDB(),\n",
"    max_steps=1,\n",
")\n",
"\n",
"agent3 = Agent(\n",
"    llm=llm,\n",
"    agent_name=\"Libertarian Leader\",\n",
"    agent_description=\"Leader of the Libertarian Party\",\n",
"    long_term_memory=ChromaDB(),\n",
"    max_steps=1,\n",
")\n",
"\n",
"# Initialize the majority voting\n",
"mv = MajorityVoting(\n",
   "    agents=[agent1, agent2, agent3],\n",
   "    output_parser=llm.majority_voting,\n",
   "    autosave=False,\n",
   "    verbose=True,\n",
   ")\n",
   "\n",
   "\n",
   "# Start the majority voting\n",
   "mv.run(\"What is your stance on healthcare?\")"
  ]
 },
 {
  "cell_type": "markdown",
  "metadata": {},
  "source": [
   "#Real-World Deployment\n"
  ]
 },
 {
  "cell_type": "markdown",
  "metadata": {},
  "source": [
   "**Multi-Agent Swarm for Logistics**\n",
   "\n",
   "Here's a production grade swarm ready for real-world deployment in a factory and logistics settings like warehouses. This swarm can automate 3 costly and inefficient workflows, safety
```

```
checks, productivity checks, and warehouse security."
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {},
   "outputs": [],
   "source": [
    "import os\n",
    "\n",
    "from dotenv import load_dotenv\n",
    "\n",
    "from swarm_models import GPT4VisionAPI\n",
    "from swarms.prompts.logistics import (\n",
    "    Efficiency_Agent_Prompt,\n",
    "    Health_Security_Agent_Prompt,\n",
    "    Productivity_Agent_Prompt,\n",
    "    Quality_Control_Agent_Prompt,\n",
    "    Safety_Agent_Prompt,\n",
    "    Security_Agent_Prompt,\n",
    "    Sustainability_Agent_Prompt,\n",
    ")\n",
    "from swarms.structs import Agent\n",
    "\n",
    "# Load ENV\n",
    "load_dotenv()\n",

    "api_key = os.getenv(\"OPENAI_API_KEY\")\n",

    "\n",

    "# GPT4VisionAPI\n",

    "llm = GPT4VisionAPI(openai_api_key=api_key)\n",

    "\n",

    "# Image for analysis\n",

    "factory_image = \"factory_image1.jpg\"\n",

    "\n",

    "# Initialize agents with respective prompts\n",

    "health_security_agent = Agent(\n",

    "    llm=llm,\n",

    "    sop=Health_Security_Agent_Prompt,\n",

    "    max_loops=1,\n",

    "    multi_modal=True,\n",

    ")\n",

    "\n",

    "# Quality control agent\n",

    "quality_control_agent = Agent(\n",

    "    llm=llm,\n",

    "    sop=Quality_Control_Agent_Prompt,\n",

    "    max_loops=1,\n",

    "    multi_modal=True,\n",

    ")\n",

    "\n",

    "\n",
    "# Productivity Agent\n",
    "productivity_agent = Agent(\n",
    "    llm=llm,\n",
    "    sop=Productivity_Agent_Prompt,\n",
    "    max_loops=1,\n",
    "    multi_modal=True,\n",
    ")\n",
    "\n",
    "# Initiailize safety agent\n",
    "safety_agent = Agent(llm=llm, sop=Safety_Agent_Prompt, max_loops=1, multi_modal=True)\n",
    "\n",
    "# Init the security agent\n",
    "security_agent = Agent(\n",
    "    llm=llm, sop=Security_Agent_Prompt, max_loops=1, multi_modal=True\n",
    ")\n",
    "\n",
    "\n",
    "# Initialize sustainability agent\n",
    "sustainability_agent = Agent(\n",
    "    llm=llm,\n",
    "    sop=Sustainability_Agent_Prompt,\n",
    "    max_loops=1,\n",
    "    multi_modal=True,\n",
    ")\n",
    "\n",
    "\n",
    "# Initialize efficincy agent\n",

    "efficiency_agent = Agent(\n",

    "    llm=llm,\n",

    "    sop=Efficiency_Agent_Prompt,\n",

    "    max_loops=1,\n",

    "    multi_modal=True,\n",

    ")\n",

    "\n",

    "# Run agents with respective tasks on the same image\n",

    "health_analysis = health_security_agent.run(\n",

    "    \"Analyze the safety of this factory\", factory_image\n",

    ")\n",

    "quality_analysis = quality_control_agent.run(\n",

    "    \"Examine product quality in the factory\", factory_image\n",

    ")\n",

    "productivity_analysis = productivity_agent.run(\n",

    "    \"Evaluate factory productivity\", factory_image\n",

    ")\n",

    "safety_analysis = safety_agent.run(\n",

    "    \"Inspect the factory's adherence to safety standards\",\n",

    "    factory_image,\n",

    ")\n",

    "security_analysis = security_agent.run(\n",

    "    \"Assess the factory's security measures and systems\",\n",

    "    factory_image,\n",

    ")\n",
        "sustainability_analysis = sustainability_agent.run(\n",
        "    \"Examine the factory's sustainability practices\", factory_image\n",
        ")\n",
        "efficiency_analysis = efficiency_agent.run(\n",
        "    \"Analyze the efficiency of the factory's manufacturing process\",\n",
        "    factory_image,\n",
        ")"
       ]
      },
      {
       "cell_type": "markdown",
       "metadata": {},
       "source": [
        "Multi Modal Autonomous Agents\n",
        "\n",
        "Run the agent with multiple modalities useful for various real-world tasks in manufacturing, logistics, and health."
       ]
      },
      {
       "cell_type": "code",
       "execution_count": null,
       "metadata": {},
       "outputs": [],
       "source": [
        "# Description: This is an example of how to use the Agent class to run a multi-modal workflow\n",
    "import os\n",

    "\n",

    "from dotenv import load_dotenv\n",

    "\n",

    "from swarm_models.gpt4_vision_api import GPT4VisionAPI\n",

    "from swarms.structs import Agent\n",

    "\n",

    "# Load the environment variables\n",

    "load_dotenv()\n",

    "\n",

    "# Get the API key from the environment\n",

    "api_key = os.environ.get(\"OPENAI_API_KEY\")\n",

    "\n",

    "# Initialize the language model\n",

    "llm = GPT4VisionAPI(\n",

    "    openai_api_key=api_key,\n",

    "    max_tokens=500,\n",

    ")\n",

    "\n",

    "# Initialize the task\n",

    "task = (\n",

    "    \"Analyze this image of an assembly line and identify any issues such as\"\n",

    "    \" misaligned parts, defects, or deviations from the standard assembly\"\n",

    "    \" process. IF there is anything unsafe in the image, explain why it is\"\n",

    "    \" unsafe and how it could be improved.\"\n",

    ")\n",
      "img = \"assembly_line.jpg\"\n",
      "\n",
      "## Initialize the workflow\n",
      "agent = Agent(\n",
      "    llm=llm, max_loops=\"auto\", autosave=True, dashboard=True, multi_modal=True\n",
      ")\n",
      "\n",
      "# Run the workflow on a task\n",
      "agent.run(task=task, img=img)"
     ]
    },
    {
     "cell_type": "markdown",
     "metadata": {},
     "source": [
      "#Build your own LLMs, Agents, and Swarms!\n"
     ]
    },
    {
     "cell_type": "markdown",
     "metadata": {},
     "source": [
      "**Swarms Compliant Model Interface**"
     ]
    },
    {
      "cell_type": "code",
      "execution_count": null,
      "metadata": {},
      "outputs": [],
      "source": [
       "from swarms import BaseLLM\n",
       "\n",
       "class vLLMLM(BaseLLM):\n",
       "    def __init__(self, model_name='default_model', tensor_parallel_size=1, *args, **kwargs):\n",
       "        super().__init__(*args, **kwargs)\n",
       "        self.model_name = model_name\n",
       "        self.tensor_parallel_size = tensor_parallel_size\n",
       "        # Add any additional initialization here\n",
       "    \n",
       "    def run(self, task: str):\n",
       "        pass\n",
       "\n",
       "# Example\n",
       "model = vLLMLM(\"mistral\")\n",
       "\n",
       "# Run the model\n",
       "out = model(\"Analyze these financial documents and summarize of them\")\n",
       "print(out)\n"
      ]
     },
     {
```

```
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "**Swarms Compliant Agent Interface**"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "from swarms import Agent\n",
  "\n",
  "\n",
  "class MyCustomAgent(Agent):\n",
  "    def __init__(self, *args, **kwargs):\n",
  "        super().__init__(*args, **kwargs)\n",
  "    \n",
  "    # Custom initialization logic\n",
  "    def custom_method(self, *args, **kwargs):\n",
  "        # Implement custom logic here\n",
  "        pass\n",
  "\n",
  "    def run(self, task, *args, **kwargs):\n",
  "        # Customize the run method\n",
    "        response = super().run(task, *args, **kwargs)\n",
    "        # Additional custom logic\n",
    "        return response\n",
    "\n",
    "# Model\n",
    "agent = MyCustomAgent()\n",
    "\n",
    "# Run the agent\n",
    "out = agent(\"Analyze and summarize these financial documents: \")\n",
    "print(out)\n"
   ]
  },
  {
   "cell_type": "markdown",
   "metadata": {},
   "source": [
    "**Compliant Interface for Multi-Agent Collaboration**"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {},
   "outputs": [],
   "source": [
    "from swarms import AutoSwarm, AutoSwarmRouter, BaseSwarm\n",
"\n",

"\n",

"# Build your own Swarm\n",

"class MySwarm(BaseSwarm):\n",

"    def __init__(self, name=\"kyegomez/myswarm\", *args, **kwargs):\n",

"        super().__init__(*args, **kwargs)\n",

"        self.name = name\n",

"\n",

"    def run(self, task: str, *args, **kwargs):\n",

"        # Add your multi-agent logic here\n",

"        # agent 1\n",

"        # agent 2\n",

"        # agent 3\n",

"        return \"output of the swarm\"\n",

"\n",

"\n",

"# Add your custom swarm to the AutoSwarmRouter\n",

"router = AutoSwarmRouter(\n",

"    swarms=[MySwarm]\n",

")\n",

"\n",

"\n",

"# Create an AutoSwarm instance\n",

"autoswarm = AutoSwarm(\n",

"    name=\"kyegomez/myswarm\",\n",

"    description=\"A simple API to build and run swarms\",\n",
```

"   verbose=True,\n",
"   router=router,\n",
")\n",
"\n",
"\n",
"# Run the AutoSwarm\n",
"autoswarm.run(\"Analyze these financial data and give me a summary\")\n",
"\n"
]
},
{
"cell_type": "markdown",
"metadata": {},
"source": [
"**``AgentRearrange``**\n",
"\n",
"Inspired by Einops and einsum, this orchestration techniques enables you to map out the relationships between various agents. For example you specify linear and sequential relationships like a -> a1 -> a2 -> a3 or concurrent relationships where the first agent will send a message to 3 agents all at once: a -> a1, a2, a3. You can customize your workflow to mix sequential and concurrent relationships"
]
},
{
"cell_type": "code",
"execution_count": null,

```
   "metadata": {},
   "outputs": [],
   "source": [
    "from swarms import Agent
from swarm_models import Anthropic, AgentRearrange \n",
    "\n",
    "## Initialize the workflow\n",
    "agent = Agent(\n",
    "    agent_name=\"t\",\n",
    "    agent_description=(\n",
    "        \"Generate a transcript for a youtube video on what swarms\"\n",
    "        \" are!\"\n",
    "    ),\n",
    "    system_prompt=(\n",
    "        \"Generate a transcript for a youtube video on what swarms\"\n",
    "        \" are!\"\n",
    "    ),\n",
    "    llm=Anthropic(),\n",
    "    max_loops=1,\n",
    "    autosave=True,\n",
    "    dashboard=False,\n",
    "    streaming_on=True,\n",
    "    verbose=True,\n",
    "    stopping_token=\"<DONE>\",\n",
    ")\n",
    "\n",
    "agent2 = Agent(\n",
    "    agent_name=\"t1\",\n",
    "    agent_description=(\n",
    "        \"Generate a transcript for a youtube video on what swarms\"\n",
    "        \" are!\"\n",
    "    ),\n",
    "    llm=Anthropic(),\n",
    "    max_loops=1,\n",
    "    system_prompt=\"Summarize the transcript\",\n",
    "    autosave=True,\n",
    "    dashboard=False,\n",
    "    streaming_on=True,\n",
    "    verbose=True,\n",
    "    stopping_token=\"<DONE>\",\n",
    ")\n",
    "\n",
    "agent3 = Agent(\n",
    "    agent_name=\"t2\",\n",
    "    agent_description=(\n",
    "        \"Generate a transcript for a youtube video on what swarms\"\n",
    "        \" are!\"\n",
    "    ),\n",
    "    llm=Anthropic(),\n",
    "    max_loops=1,\n",
    "    system_prompt=\"Finalize the transcript\",\n",
    "    autosave=True,\n",
    "    dashboard=False,\n",
    "    streaming_on=True,\n",
    "    verbose=True,\n",
    "    stopping_token=\"<DONE>\",\n",
    ")\n",
    "\n",
    "\n",
    "# Rearrange the agents\n",
    "rearrange = AgentRearrange(\n",
    "    agents=[agent, agent2, agent3],\n",
    "    verbose=True,\n",
    "    # custom_prompt=\"Summarize the transcript\",\n",
    ")\n",
    "\n",
    "# Run the workflow on a task\n",
    "results = rearrange(\n",
    "    # pattern=\"t -> t1, t2 -> t2\",\n",
    "    pattern=\"t -> t1 -> t2\",\n",
    "    default_task=(\n",
    "        \"Generate a transcript for a YouTube video on what swarms\"\n",
    "        \" are!\"\n",
    "    ),\n",
    "    t=\"Generate a transcript for a YouTube video on what swarms are!\",\n",
    "    # t2=\"Summarize the transcript\",\n",
    "    # t3=\"Finalize the transcript\",\n",
    ")\n",
        "# print(results)\n",
        "\n"
       ]
      }
     ],
     "metadata": {
      "accelerator": "GPU",
      "colab": {
       "gpuType": "T4",
       "private_outputs": true,
       "provenance": []
      },
      "kernelspec": {
       "display_name": "Python 3",
       "name": "python3"
      },
      "language_info": {
       "name": "python"
      }
     },
     "nbformat": 4,
     "nbformat_minor": 0
    }
```