卒業研究科目「ゼミナール 田中」
第2回 第1章
Pythonの文法項目

1

いちばんゼミナールらしくない回

教員がマイクに向かってしゃべり続けるのは今回まで

2

第2回の内容

- 第1章 Pythonの文法項目
  - リスト
  - 辞書型
  - 関数定義、呼び出し
  - その他
- 第2章プログラムそのものではないが、大事なこと
  - GitHubリポジトリ
  - 環境変数
  - その他
- 第3章 チャットAPI
- 小テスト

開発に必要なPythonの基礎を復習し、プログラムを実行できる

GOAL

3

今からPythonの基礎を復習？

1. Python に特有の書き方、出力に馴染めるようになろう
2. API活用で頻出の書き方にも馴染めるようになろう

教科書によく出てくる文法項目

4

(1) 関数の定義

```
def greet():
    print("こんにちは！")
```

- def を使って関数を定義できる（define）
- greet は「こんにちは！」と表示する関数の名前である
- よく使う処理を関数にまとめることで再利用しやすくなる

5

(2) 関数の呼び出し

```
def greet():
    print("こんにちは！")

greet()
```

- 定義した関数は関数名のあとに () をつけて呼び出す
- greet() を実行すると関数内の処理がそのまま実行される
- greet() の呼び出し箇所はインデントされていないので、その直前で関数定義は終わっている

6

(3) 関数からのreturn

```
def get_greeting():
    return "こんにちは！"

message = get_greeting()
print(message)
```

- return で実行結果を呼び出し元に返すことができる
- get_greeting() が返した「こんにちは！」は message に代入される
- 関数の中で計算した結果や値を使い回すときに return を使う

7

(4) リストの作成

```
fruits = ["りんご", "バナナ", "みかん"]
```

- リストは `[]` を使って作成できる
- 複数の値を順番に並べて管理できるデータ型である
- 上記の例では「りんご」「バナナ」「みかん」が順番に格納されている

8

(5) リストとappend

```
fruits = ["りんご", "バナナ", "みかん"]
fruits.append("ぶどう")
print(fruits)
```

- `append()` を使うとリストの末尾に要素を追加できる
- この例では「ぶどう」がリストの最後に追加される
- リストを操作する基本的な方法であり、要素を動的に追加したいときに使う
- （参考）最後ではない任意の位置に追加するなら insert()

9

(6) joinで文字列を作る

```
fruits = ["りんご", "バナナ", "みかん"]
result = "-".join(fruits)
print(result)
```

- `join()` はリストの要素を指定した区切り文字で結合する
- この例ではリスト内の要素を「-」で繋げた文字列が作られる
- 文字列の結合に便利な方法であり、リスト内の要素を1つの文字列として扱いたいときに使う

10

(7) f文字列

```
name = "太郎"
greeting = f"こんにちは、{name} さん！"
print(greeting)
```

- f文字列（フォーマット済み文字列リテラル）は、文字列内に変数を埋め込む方法である
- `f` を使って文字列の中に `{}` を使って変数や式を埋め込むことができる
- コードが簡潔になり、可読性が向上するため便利

11

forループ

```
fruits = ["りんご", "バナナ", "みかん"]

for fruit in fruits:
    print(fruit)
```

- `for` ループを使うと、リストなどの要素を1つずつ繰り返し処理できる
- この例では、リストの各要素（果物）を順番に取り出して `print` で表示している
- 反復処理に最も基本的でよく使用される構文

12

ifでの条件分岐

```
age = 20
if age >= 18:
    print ("成人です")
else:
    print ("未成年です")
```

- `if` 文を使うと、条件に応じた処理を選択できる
- 条件式が `True` ならその処理を実行し、`False` の場合は `else` 節が実行される
- 分岐処理を使って、プログラムの動作を条件によって変えることができる

13

辞書の定義と操作

```
person = {"name": "太郎", "age": 20}

print (person ["name"] )
person ["age"] = 21
```

- キーと値のペアを格納するデータ型
- `{}` を使って定義し、キーを使って値にアクセスすることができる
- この例では `person ["name"]` で名前にアクセスし、年齢を `person ["age"]` で更新している
- 教科書中ではプロンプトの構築で利用

14

JSONの読込

```
import json

json_data = '{"name": "太郎", "age": 25, "city": "東京"}'  # JSON文字列

data = json.loads (json_data)  # JSON文字列をPythonの辞書に変換

print (data)
print (data ["name"] )
```

- `json` モジュールは、JSONデータをPythonのデータ型に変換するために使う
- `json.loads ()` でJSON形式の文字列をPythonの辞書(`dict`)に変換する
- 変換後は、通常の辞書と同じようにアクセスできる

15

クラスの定義

```
class Person:
    def __init__ (self, name, age) :
        self.name = name
        self.age = age

person = Person ("太郎", 20)
print (person.name)
```

- `class` を使ってクラスを定義し、オブジェクト(インスタンス)を作ることができる
- クラスは属性(変数)とメソッド(関数)をまとめたもの
- `__init__ ()` はクラスがインスタンス化されるときに最初に呼ばれる初期化メソッドである

16

ラムダ関数

```
square = lambda x: x ** 2
print (square (4) )
```

- 無名の関数を定義するための方法
- `lambda` を使い、引数と処理を1行で定義することができる
- シンプルな処理や一時的な関数定義に便利

17

リスト内包表記

```
numbers = [1, 2, 3, 4]
squares = [x ** 2 for x in numbers]

print (squares)
```

- リストを簡潔に作成できる
- `[式 for x in iterable]` の形式で、iterableの各要素に対して処理を施して新しいリストを作る
- Iterableとは、反復して処理できるオブジェクト
- コードが簡潔になり、可読性が向上する

18

オブジェクトのドット表記

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def start_engine(self):
        print(f"{self.brand} {self.model} のエンジンが始動しました")

my_car = Car("トヨタ", "カローラ") # インスタンス化

print(my_car.brand) # ドット表記で属性にアクセス
my_car.start_engine() # ドット表記でメソッドにアクセス
```

- オブジェクトの属性やメソッドにアクセスするための方法
- この例では、`my_car` は `Car` クラスのインスタンスであり、`my_car.brand` で属性 `brand` にアクセスし、`my_car.start_engine()` でメソッド `start_engine` を呼び出している
- 教科書では、APIの利用で頻出

19

まとめ

- 関数の定義(def)
- 関数の呼び出し(関数名())
- 関数からの戻り値(return)
- リストの作成([])
- リストへの要素追加(append)
- 文字列の結合(join)
- 文字列フォーマット(f文字列)
- 繰り返し処理(forループ)
- 条件分岐(if文)
- 辞書の定義と操作({"key": value})
- JSON
- クラスの定義(class)
- ラムダ関数(lambda)
- リスト内包表記([式 for x in iterable])
- オブジェクトのドット表記(オブジェクト.属性やメソッド)

いずれも、教科書に現れる文法項目

20

Cyber University

第2回 第1章 おわり

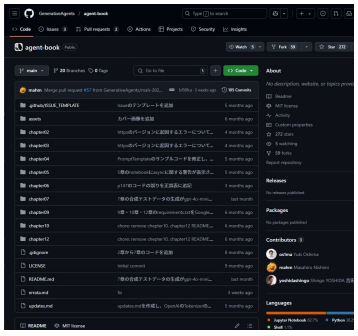
21

Cyber University

卒業研究科目「ゼミナール 田中」
第2回 第2章
プログラムそのものではないが、大事なこと

22

教科書著者グループのGitHubリポジトリ



- たいへん親切な作り
- 通常なら…
  - git cloneする
  - Zipダウンロードする
  - 1件ずつノートブックをダウンロードする
  - コピペする

https://github.com/GenerativeAgents/agent-book

23

ボタン1発でGoogle Colabに行ける

- 行った後は「ドライブにコピー」する
- コピーすれば、自分のGoogleドライブから開ける
- コピーしないと…
  - 作業内容は失われる
  - 他者との共有や共同編集ができない
- ノートブックのファイル名は、わかりやすいものに変えておく（第2章notebook.ipynb など）

24

パッケージのインストール

```
[ ]  !pip install tiktoken==0.7.0
```

- 上の例では
  - パッケージの名前がtiktoken
  - バージョン番号が0.7.0
- バージョン番号は指定する癖をつけたい
- バージョンが合っていないと「教科書での説明と挙動が合わなくなる」「エラーで止まる」が発生する

25

インストールできているか不安な場合

```
!pip show tiktoken
```

```
!pip list
```

- showで、指定したパッケージの詳細情報が出る
- listで、導入済パッケージの一覧が出る

26

モジュールのインポート

```
import tiktoken

text = "ChatGPT"
#以下略
```

- importの時点でエラーが出たら?
  - インストールに失敗している
  - 綴りの間違い

27

環境変数の設定

```
os.environ["OPENAI_API_KEY"]
```

- 上の例の場合、環境変数"OPENAI_API_KEY"に予め値を入れておかないといけない
- 値がないと SecretNotFoundError: Secret OPENAI_API_KEY does not exist.
- Colabのシークレット機能でキーを保存する方法は、教科書pp.24-25を見てください

28

プログラム中での環境変数の取得

```
import os
from google.colab import userdata

os.environ["OPENAI_API_KEY"] = userdata.get("OPENAI_API_KEY")
```

- シークレットに保存しておいた値は、キーを指定して userdata.get () で取り出す

29

なぜシークレットで環境変数に入れるのか?

```
# 直接設定する方法…やらないでください
os.environ["OPENAI_API_KEY"] = "私のキーの値はこれです!"
```

- Colabのノートブック内に平文でキーの値を書き込むと、漏洩につながる
- (間違えてノートブックの共有や公開をして) 人に見られると、従量課金のあなたのお金を人に使われてしまう!

30

まとめ

- 教科書GitHubリポジトリへのアクセス方法
- パッケージのインストール方法
- インストール済パッケージの確認方法
- 環境変数の設定方法
- APIキーはシークレットで環境変数に入れる
- キーの値を環境変数から取り出してプログラム中で取り出す方法

31

Cyber University

第2回 第2章 おわり

32

Cyber University

卒業研究科目「ゼミナール 田中」
第2回 第3章
OpenAIのチャットAPI

33

(再) 教科書とスケジュールの対応

- 第1章 LLMアプリケーション開発の基礎
- 第2章 OpenAIのチャットAPIの基礎
- 第3章 プロンプトエンジニアリング
- 第4章 LangChainの基礎
- 第5章 LangChain Expression Language(LCEL)徹底解説
- 第6章 Advanced RAG
- 第7章 LangSmithを使ったRAGアプリケーションの評価

第3回～6回
シラバスの「プロンプトエンジニアリングとRAG編」

第7回～10回
「AIエージェント導入編」

- 第8章 AIエージェントとは
- 第9章 LangGraphで作るAIエージェント実践入門
- 第10章 要件定義書生成AIエージェントの開発
- 第11章 エージェントデザインパターン
- 第12章 LangChain/LangGraphで実装するエージェントデザインパターン

第11回～13回
「デザインパターン編」

34

(再) 教科書とスケジュールの対応

- 第1章 LLMアプリケーション開発の基礎
- 第2章 OpenAIのチャットAPIの基礎
- 第3章 プロンプトエンジニアリング
- 第4章 LangChainの基礎
- 第5章 LangChain Expression Language(LCEL)徹底解説
- 第6章 Advanced RAG
- 第7章 LangSmithを使ったRAGアプリケーションの評価

第3回～6回
シラバスの「プロンプトエンジニアリングとRAG編」

第7回～10回
「AIエージェント導入編」

- 第8章 AIエージェントとは
- 第9章 LangGraphで作るAIエージェント実践入門
- 第10章 要件定義書生成AIエージェントの開発
- 第11章 エージェントデザインパターン
- 第12章 LangChain/LangGraphで実装するエージェントデザインパターン

第11回～13回
「デザインパターン編」

35

教科書第2章は輪講(発表)の対象外

- プログラムの例があるのに、なぜ対象外?
  - 本題は、教科書のタイトルにある通り「LangChainとLangGraph」
  - 第2章はLangChainやLangGraphのラッパーに包まれていない、素のままのOpenAIのAPIの使い方
  - 本題から外れているので対象外にしたが、見ておいて損はない
  - LangChainやLangGraphが素のOpenAIのと比べて何が嬉しいか、がわかる
- この章で、教科書第2章のプログラム実行の概要だけお見せします

36

Chat Completions API の呼び出し

```
from openai import OpenAI

client = OpenAI()

response = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=[
        {"role": "system", "content": "あなたはとっても優秀なアシスタント。"},
        {"role": "user", "content": "こんにちは！私はジョンと言います！"},
    ],
)
print(response.to_json(indent=2))
```

- モデル名を指定
- systemはモデルの動作や応答のガイドライン
- userは実際のユーザーが入力したメッセージ
- JSON形式で整形して表示させる

37


```
{
  "id": "chatcmpl-BKau6jiNLbTVZA5AshQqeA8SSisSC",
  "choices": [
    {
      "finish_reason": "stop",
      "index": 0,
      "logprobs": null,
      "message": {
        "content": "こんにちは、ジョンさん！お会いできて嬉しいです。今日はどんなことをお話ししましょうか？",
        "refusal": null,
        "role": "assistant",
        "annotations": []
      }
    }
  ],
  "created": 1744248118,
  "model": "gpt-4o-mini-2024-07-18",
  "object": "chat.completion",
  "service_tier": "default",
  "system_fingerprint": "fp_44added55e",
  "usage": {
    "completion_tokens": 28,
    "prompt_tokens": 32,
    "total_tokens": 60,
    "prompt_tokens_details": {
      "cached_tokens": 0,
      "audio_tokens": 0
    },
    "completion_tokens_details": {
      "reasoning_tokens": 0,
      "audio_tokens": 0,
      "accepted_prediction_tokens": 0,
      "rejected_prediction_tokens": 0
    }
  }
}
```


「こんにちは」のような応答だけでなく、付随するメタデータも得られる

38

会話履歴を踏まえた応答を得る

```
response = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=[
        {"role": "system", "content": "あなたはとっても優秀なアシスタント。"},
        {"role": "user", "content": "こんにちは！私はジョンと言います！"},
        {"role": "assistant", "content": "こんにちは、ジョンさん！お会いできて嬉しいです。今日はどんなことをお話ししましょうか？"},
        {"role": "user", "content": "私の名前が分かりますか？"},
    ],
)
print(response.to_json(indent=2))
```

- このAPIはステートレス
- 『こんにちは、ジョンさん！』の行に注目
- 過去の履歴を使いたいなら、リクエストに含める

39


```
{
  "id": "chatcmpl-BKb12AuT1qdrHoSzxvlUAmwFk4eQn",
  "choices": [
    {
      "finish_reason": "stop",
      "index": 0,
      "logprobs": null,
      "message": {
        "content": "はい、ジョンさんですね！他にお話ししたいことがあれば教えてください。",
        "refusal": null,
        "role": "assistant",
        "annotations": []
      }
    }
  ],
  "created": 1744248548,
  "model": "gpt-4o-mini-2024-07-18",
  "object": "chat.completion",
  "service_tier": "default",
  "system_fingerprint": "fp_44added55e",
  "usage": {
    "completion_tokens": 23,
    "prompt_tokens": 76,
    "total_tokens": 99,
    "prompt_tokens_details": {
      "cached_tokens": 0,
      "audio_tokens": 0
    },
    "completion_tokens_details": {
      "reasoning_tokens": 0,
      "audio_tokens": 0,
      "accepted_prediction_tokens": 0,
      "rejected_prediction_tokens": 0
    }
  }
}
```


履歴をリクエストに入れたので「ジョンさんですね」と返せた

40

ストリーミングで応答を得る

```
response = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=[
        {"role": "system", "content": "あなたはとっても優秀なアシスタント。"},
        {"role": "user", "content": "こんにちは！私はジョンと言います！"},
    ],
    stream=True,
)

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

- 結果が少しずつ表示され、即時性が高い応答を得られる
- ChatGPTでのWeb画面での表示に近い
- 前の例のような、JSONでの応答表示ではない
- （応答は省略）

41

JSON モード

```
from openai import OpenAI

client = OpenAI()

response = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=[
        {
            "role": "system",
            "content": "人物一覧を次のJSON形式で出力してください。¥n{"people": ["aaa", "bbb"]}",
        },
        {
            "role": "user",
            "content": "昔々あるところにおじいさんとおばあさんがいました",
        },
    ],
    response_format={"type": "json_object"},
)
print(response.choices[0].message.content)
```

- 応答を人間ではなくアプリケーションで使いたい時
- response_format={“type”: “json_object”} とする

42

Vision（画像入力）

```
from openai import OpenAI
client = OpenAI()
image_url = "https://raw.githubusercontent.com/yoshidashingo/langchain-book/main/assets/cover.jpg"

response = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=[
        {
            "role": "user",
            "content": [
                {"type": "text", "text": "画像を説明してください。"},
                {"type": "image_url", "image_url": {"url": image_url}},
            ],
        }
    ],
)
print(response.choices[0].message.content)
```

- マルチモーダルAI
- 上の例では、教科書の表紙の画像について説明させている

Function Calling

```
import json
def get_current_weather(location, unit="fahrenheit"):
    if "tokyo" in location.lower():
        return json.dumps({"location": "Tokyo", "temperature": "10", "unit": unit})
    elif "san francisco" in location.lower():
        return json.dumps(
            {"location": "San Francisco", "temperature": "72", "unit": unit}
        )
    elif "paris" in location.lower():
        return json.dumps({"location": "Paris", "temperature": "22", "unit": unit})
    else:
        return json.dumps({"location": location, "temperature": "unknown"})
# 以下、省略します
```

- LLMはPythonのコードを直接実行はできない
- LLMに「この関数を使いたいです」という応答を返させる
- 応答が来たら、（人間の手元の）Pythonの実行環境で関数を実行する
- 単なるチャットのやり取りを超え「LLMによる自然言語での応答と、人間が書いたプログラムのハイブリッド利用」へと踏み出す第一歩

まとめ

- 教科書第2章は本題から外れているので対象外だが、見ておいて損はない
- Chat Completions API
- 会話の履歴を踏まえさせる
- ストリーミングで応答
- JSONモード
- 画像入力でマルチモーダルAI
- 人間が書いたプログラムと融合させるFunction Calling

Cyber University

第2回 第3章 おわり