<script type="module"> import mermaid from 'https://cdn.jsdelivr.net/npm/mermaid@11.4.1/dist/mermaid.esm.min.mjs'; mermaid.initialize({ startOnLoad: true }); </script>

データサイエンティストのための

ソフトウェア工学入門

目次

1. ソフトウェア工学の重要性

```
mindmap
  root((ソフトウェア工学 ))
    車輪の再発明を防ぐ
      既存ライブラリの活用
      ベストプラクティス
      設計パターン
    巨人の肩に乗る
      オープンソース
      コミュニティ
      ドキュメント
    効率的な開発
      チーム開発
      品質管理
      保守性
```

1.1 なぜソフトウェア工学が必要か？

データサイエンティストにとって、ソフトウェア工学の知識が重要な理由：

- **再現性の確保**
  - 分析結果の再現
  - 他者との共同作業
- **保守性の向上**
  - 6 ヶ月後の自分が理解できるコード
  - チームでの開発効率向上

1.2 車輪の再発明を防ぐ

既存ライブラリの活用

欲しい機能は既にある可能性が高い

機能の再開発に時間をかける必要がない

バグが発生する可能性が低い

1.3 巨人の肩に乗る

オープンソース

先人たちの知見を活用

保守性が高い

機能拡張

自分では思いつかないような機能を利用できる

人生は有限

2. 用法用量を守って

高度化するソフトウェア開発の手法を利用することは重要ですが、その利用にはユースケースに応じた手法を選択することが重要です。

2.1 高度化するソフトウェア開発

```
mindmap
  root((□□□□□□□□□))
    □□□□
      □□□□□
      □□□□□
      □□□□□
    □□□□□□
      □□□□
      □□
      □□
      □□□
      □□□□
    □□□□
      □□□□□□□
      □□□□□
      □□□□□□□□□□□□
      □□□□□□
    □□□□□□
      □□□□
      □□□□□□□
      □□□□□
      □□□□□□□
```

2.2 適切なコーディング手法の選択

```c
// メモリー割当
#include <stdlib.h>

void memory_leak_example() {
    int *ptr = (int *)malloc(10 * sizeof(int));
    // メモリーリーク : mallocで確保したメモリが解放 (free)されていない
}
```

2.3 帰納と演算

帰納法は個別の事例や観察から一般的な法則や結論を導き出す思考方法です。
例えば「今まで見たカラスは全て黒かった → カラスは黒い」というような推論です。多くの具体例から共通点を見出し、一般化するプロセスです。

演繹法は一般的な法則や前提から個別の結論を導き出す思考方法です。例えば「全ての哺乳類は心臓を持つ。犬は哺乳類である。→ 従って犬は心臓を持つ」というような推論です。

- 思考の方向性
  - 帰納法: 個別 → 一般
  - 演繹法: 一般 → 個別
- 確実性
  - 帰納法: 結論は確率的・蓋然的
  - 演繹法: 前提が正しければ結論は必然的に正しい

帰納的アプローチ

帰納的アプローチは、**具体的な事例や観察から一般的な法則や原理を導き出す手法**です。これは、個々のケースを分析し、それらの共通点やパターンを見つけ出すことで、より広範な結論を得ることを目指します。

例えば、データサイエンスの分野では、様々なデータセットを分析して、その中で共通するパターンや傾向を見つけ出すことが多くあります。このような分析を通じて、新しいデータセットに対しても同様の傾向があると推測することができます。

帰納的アプローチの利点は、実際のデータや経験に基づいているため、現実的で実践的な解決策を提供できる点です。しかし、観察された事例が限られている場合や、偏りがある場合には、導き出された結論が必ずしも正確でない可能性があるため、注意が必要です。

帰納的アプローチの例

```python
import pandas as pd
import numpy as np
from typing import Dict

def calculate_metrics(data: pd.DataFrame) -> Dict[str, float]:
    """月次売上データの基本的な統計量と傾向を計算する
    """
    return {
        # 基本統計量
        'sales_mean': data['sales'].mean(),
        'sales_median': data['sales'].median(),
        'sales_std': data['sales'].std(),

        # トレンド（前年同月比）
        'yoy_growth': (data['sales'].iloc[-1] / data['sales'].iloc[-13] - 1) * 100,

        # 季節性（月別平均）
        'monthly_avg': data.groupby(data['date'].dt.month)['sales'].mean().to_dict()
    }

})
```

2.3 帰納と演算

演繹的アプローチ

演繹的アプローチは、**一般的な原理や法則から具体的な結論や予測を導き出す手法**です。これは、既存の理論や原理を前提として、それらを論理的に推論することで、新しい結論を得ることを目指します。

例えば、特定の設計パターンが成功を収めていることを観察した場合、その設計パターンが他のプロジェクトでも有効であると推測することができます。このようにして得られた知見は、新しいプロジェクトにおける意思決定や戦略の策定に役立ちます。

ソフトウェア開発においては、演繹的アプローチは一般的な手法であり、原理原則を絶対のものとして最適な設計や実装を行っていきます。

演繹的アプローチの例

```python
# FastAPI のサンプルコード
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def read_root():
    """
    2つの数値を足し合わせて返す
    """
    a = 1
    b = 2
    sum_of_a_and_b = a + b
    return {"result": sum_of_a_and_b}
```

2.4 個人とチームと消費期限

ソフトウェア開発においては通常、複数人での開発が行われます。
そのため、**「自分だけが分かれば良いコード」ではなく、「チームでも分かるコード」** を作成することが重要です。

この点でもデータサイエンスにおけるコーディングとは異なる点があると思います。
多くのケースではデータサイエンティストは一人でコーディングを行うことが多いため、再利用性やコメントの重要性は低いものになっているかと思います。
また、成果物であるコードについても、保守性やメンテンナンス性を高く保つ必要があるケースは稀であるため、コードの可読性、保守性、再利用性は低いものであっても構いません。

ただ、**「1 ヶ月後の自分は他人」** という認識を持つことで、未来の自分が困らないコーディングを行うことはとても重要なものであると考えています。

3. 効果的なコーディング手法

開発効率を上げる代表的なコーディング手法を紹介します。
より高度なコーディング手法は扱うプログラミング言語やユースケースによって異なるため、ここでは紹介しません。
初歩的ではありますが、コーディングの質を高めるコメントと可読性について紹介します。

3.1 コメントの重要性

```
# 悪いコメントの例
def calculate_metrics(data: pd.DataFrame) -> Dict[str, float]:
    # データをクリッピングする
    cleaned_data = clip_outliers(data, percentile=99)
    # 統計量を計算する
    return compute_statistics(cleaned_data)
```

```python
# 良いコメントの例
def calculate_metrics(data: pd.DataFrame) -> Dict[str, float]:
    """データセットの基本的な統計量を計算する

    Args:
        data: 分析対象のデータフレーム
    Returns:
        統計量の辞書（平均、中央値、標準偏差など）
    """
    # 異常値の影響を軽減するため、99パーセンタイルでクリッピング
    cleaned_data = clip_outliers(data, percentile=99)
    return compute_statistics(cleaned_data)
```

3.2 可読性の高いコード作成

コードは書く時間よりも読む時間の方が圧倒的に多いということを頭に入れておくことが重要です。
また、同じ処理結果でも可読性の高いコードにすると実行したかった処理のロジックが整理され、結果的に開発効率の向上やバグの発生を防ぐことができます。
以下のコードは、データフレームに対して複数の操作を再帰的に適用するコードの例です。

```
# □□□□□□□□□□□□
def proc(d,ops,i):
 if i>=len(ops):return d
 return proc(ops[i](d),ops,i+1)
ops=[lambda x:x.assign(sepal_length=lambda d:d['sepal_length']+d['sepal_width']),lambda x:x.query("sepal_length>8")]
df_rev=proc(df,ops,0)
```

同じ処理でも可読性の高いコードにすると、実行したかった処理のロジックが整理され、結果的に開発効率の向上やバグの発生を防ぐことができます。
以下の例は同じ処理をメソッドチェーンで実行したものです。

```
#  可読性の高いコードの例
df_processed = (
    df
    # sepal_lengthとsepal_widthを足した新しい列を作成
    .assign(
        sepal_length=lambda x: x['sepal_length'] + x['sepal_width']
    )
    #  条件に合うデータを抽出
    .query("sepal_length > 8")
)
```

メソッドチェーンとは？

メソッドチェーン（Method Chaining）とは、オブジェクト指向プログラミングにおいて、複数のメソッドを連続して呼び出すことで、一連の操作を効率的かつ直感的に記述する手法です。

1. **可読性の向上**
    - 処理の流れが視覚的に分かりやすくなります。
    - 各処理が明確に区切られ、理解しやすくなります。
2. **コードの簡潔化**
    - 複数の操作を一連の流れとして記述できるため、コードが短くなります。
    - 不要な中間変数を減らすことができます。
3. **メンテナンスの容易さ**
    - 各メソッドが独立しているため、追加や修正が容易です。
    - 処理の順序を簡単に変更できます。

4. 開発環境の構築と活用

開発効率を高めるためには、開発環境の構築と活用が肝となります。
代表的な Editor である VSCode と Jupyter Notebook の組み合わせを紹介します。
また、バージョン管理や環境差分を管理する devcontainer についても紹介します。

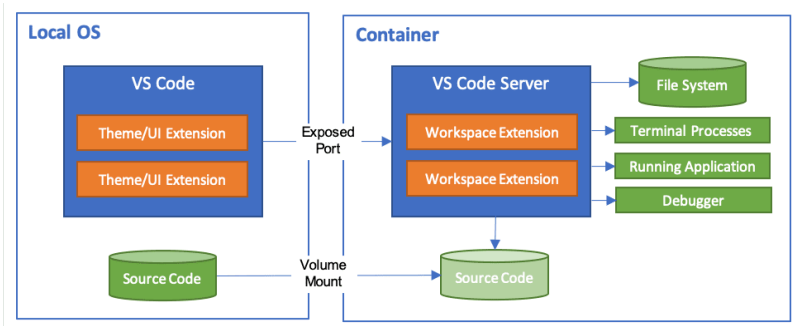
4.1 VSCode と Jupyter Notebook

VSCode とは、Microsoft が開発した**統合開発環境（IDE）** です。
この VSCode の拡張機能である **Jupyter Notebook プラグイン** を利用することで、Jupyter Notebook のようにコードを実行できるようになります。
後述する Docker コンテナを利用して、VSCode と Jupyter Notebook の環境を統一することができるため、チーム開発においても**効率的な開発**が可能になります。

Visual Studio Code

4.2 devcontainer による環境統一

devcontainer とは、Docker コンテナを利用して、開発環境を統一することができるツールです。
VSCode の拡張機能である **Remote - Containers** を利用することで、VSCode の中で Docker コンテナを利用することができるようになります。



.devcontainer > devcontainer.json > {} customizations > {} vscode > [] extensions

```jsonc
{
    "name": "Python 3 for Jupyter Notebook",
    // Or use a Dockerfile or Docker Compose file. More info: https://containers.dev/guide/dockerfile
    "image": "mcr.microsoft.com/devcontainers/python:1-3.12-bullseye",
    // Features to add to the dev container. More info: https://containers.dev/features.
    // "features": {},
    "customizations": {
        // Configure properties specific to VS Code.
        "vscode": {
            "settings": {},
            "extensions": [
                "ms-azuretools.vscode-docker",
                "streetsidesoftware.code-spell-checker",
                "Gruntfuggly.todo-tree",
                "oderwat.indent-rainbow",
                "PKief.material-icon-theme",
                "mosapride.zenkaku",
                "ms-python.vscode-pylance",
                "ms-python.python",
                "KevinRose.vsc-python-indent",
                "janisdd.vscode-edit-csv"
            ]
        }
    },
    // Use 'forwardPorts' to make a list of ports inside the container available locally.
    // "forwardPorts": [],
    // Use 'postCreateCommand' to run commands after the container is created.
    "postCreateCommand": "pip3 install --user -r requirements.txt"
    // Configure tool-specific properties.
    // "customizations": {},
    // Uncomment to connect as root instead. More info: https://aka.ms/dev-containers-non-root.
    // "remoteUser": "root"
}
```

5. AI ツールの活用

現在のソフトウェア開発においては、AI ツールの活用が不可欠です。
AI ツールを活用することで、開発効率の向上やバグの発生を防ぐことができます。
データサイエンスにおいては Python や R を活用した成果物が多く、またその規模も数百行のコードで構成されるものが多いため、生成 AI との親和性が非常に高いものになります。
本章では代表的な AI ツールである GitHub Copilot と Cursor の活用方法を紹介します。

5.1 GitHub Copilot の活用

GitHub Copilot は、GitHub が提供する AI ツールです。
GitHub Copilot は、コードの補完やリファクタリング、エラー検出などの機能を提供しています。

GitHub Copilot の公式サイト

5.2 Cursor の活用

Cursor は、Anysphere が提供する AI ファーストのエディタです。
OSS である VSCode をフォークして開発されているため、VSCode の殆どの機能、拡張プラグインが利用できます。
VSCode を独自に拡張開発しているため、AI の利用を前提とした機能が多数存在します。

Cursor の公式サイト
AI 搭載エディタ Cursor の紹介と機械学習コンペでの使用レビュー

まとめ

- ソフトウェア工学の基礎を理解する
- 用途によって必要とされるコーディングは異なることを理解する
- 適切なツールと手法を選択する
- AI ツールを効果的に活用する

参考資料・Q&A

参考資料

- Docker の公式サイト
- VSCode Documentation
- GitHub Copilot
- Cursor
- AI 搭載エディタ Cursor の紹介と機械学習コンペでの使用レビュー

参考文献

- リーダブルコード ―より良いコードを書くためのシンプルで実践的なテクニック
- プリンシプル オブ プログラミング 3 年目までに身につけたい一生役立つ 101 の原理原則
- 退屈なことは Python にやらせよう 第 2 版 ―ノンプログラマーにもできる自動化処理プログラミン
- 入門 Python 3 第 2 版
- 独学プログラマー Python 言語の基本から仕事のやり方まで
- Python で学ぶあたらしい統計学の教科書 第 2 版
- Python で動かして学ぶ！あたらしい機械学習の教科書 第 3 版
- 図解! Docker のツボとコツがゼッタイにわかる本

最後に補足 (Marp について)

今回の資料は Marp を利用して作成しました。
Marp とは Markdown 形式で PowerPoint や PDF を作成できるツールです。

以下のサイトと Public にアップロードした本資料のコードを参考にし、資料作成に活用してください。

- Marp の公式サイト
- Public においた本資料のコード