**Leitpfaden – Coding:**

* **General Aspects:**
  + **Comments**
    - Ziele:
      1. Beschreibung der Motivation des Codes  
         („Erklärende Comments“, keine reine Code-Beschreibung)
      2. Erläuterung einzelner Variable
      3. Gliederung des Codes
      4. „Lessons Learned“ mit Informationen für die zukünftige Instandhaltung
    - Arten:
      1. „Inline Comments“:
         * Hinter der Zeile – beginnend mit zwei Leerzeichen
         * Example:   
           a = np.array([0.5,0.5]) # Weights of PF for Optimization
      2. Block-Kommentar:
         * Längerer Kommentar der mehrere Zeilen umfasst.
         * Startet in jeder Zeile mit „# “
         * Example
    - No Gos:
      1. Professionell bleiben (Sprachgebrauch, etc.)
      2. Nicht die Zeilengrenze übersch
      3. Keine reine Wiederholung der Code-Operationen.  
         [Fokus auf die Motivation und das Resultat]
      4. ToDo Kommentare begrenzen,   
         Besser Codetags in Gitlab nutzen
  + **Static Codeanalysis**
  + **Dynamische Code-Analyse**

* **Functions:**
  + Naming
    - Durch den Namen muss klar sein, was die Funktion tut.
    - Example:
  + Naming - Format
    - small letters
    - snake\_style
    - Example:
  + Annotations / Typhinweise:
    - Arguments: Format and Dimension
    - Output: Format and Dimension
    - Example:
  + **Umfang der Funktionen:**
    - Kompromiss zwischen Intra-Funktionskomplexität und Inter-Funktionskomplexität
    - Zu kurz wie möglich, aber…
    - Keine Vollständige „Aufsplittung“ in Unterfunktionen
    - Ausgliederung in Unterfunktionen bei der Vermutung, dass diese auch für andere Aspekte / Funktionen nützlich sein können.
    - Ideal: 30 Zeilen, Max. 100 Zeilen
  + **Docstring:**
    - Alle Klassen, Funktionen, Methoden sollten einen Docstring haben.
    - Ziel:
      1. 1-2 Sätze zum Ziel / Motivation des Ansatzes
    - Parameter
      1. Kurze Beschreibung der Parameter  
         Dimension + Inhalt + Datentyp
    - Returns:
      1. Kurze Beschreibung der Parameter  
         Dimension + Inhalt + Datentyp
    - Notes:
      1. Optional
      2. Kurzes Beschreiben von Besonderheiten
  + **Initial Dimension / Consistency Checks**
    - Starting Point should be a Check of the Dimensions of the Inputs   
      (if necessary, esp. when Matrix Multiplication is used afterwards)
  + Eindeutige Datentyp der RÜckgabe:
    - Eine Funktion soll möglichst immer denselben Datentyp zurückgeben
  + **Debugging:**
    - Keine „None“ für Fehlerrückgabe
    - Ausnahmen bei Fehlern zurückgeben und keine Fehlercodes
* **Module:**
  + **Docstring**
    - **Kurze Beschreibung des Modulinhalts**
* **Testing:**
  + **Jedes Modul ein Set haben aus:**
    - **1) Input**
    - **2) Ergebnis**
  + **Notwendig für Testing und Unit-Tests**
  + **Ggf. Trennung: 1D / 2D Matrix und Spezialfälle**

**Einsatz in der PROD / PF Optimierung etc.:**

* **Vorheriger Durchlauf mit aktuellen Environment von spezifischen Input zum Testen ob ein spezifischer Output generiert wird,**

**Mehrfach ausführen einzelner zeilen darf nicht das Gesamtergebnis ändern**

**Gitlab – Aufbau:**

* Offene Fragen:
  + Fixe Unterteilung in „Develop“ / „Test“ / „Prod“ Branch?
  + Fixe Unterteilung in Test / Prod

Alle Entwicklungen sind mit Nebenbranches (mit Basis der PROD)

https://nvie.com/posts/a-successful-git-branching-model/

* Änderungen am VENV im Zeitablauf:
  + Verwendung von requirements.txt
  + Verwendung von CI / CD Pipelines mit Unit- und Integrationstests
  +  **Separate Umgebungen für alte und neue Pakete:** Behalten Sie eine funktionierende Version des alten VENV und vergleichen Sie das Verhalten des Codes mit der neuen Version. So können Sie Probleme isolieren.
* Code-Testing:
  + Unit Tests für automatische Überprüfung der Funktionen
  + Statische Codeanalyse
  + Dynamische Codeanalyse
* Tags und Issues für Fehler identicikation

Um eine IT-Infrastruktur in GitLab mit einer klaren Aufteilung in Entwicklungs-, Test- und Produktionsumgebungen sowie der Verwendung von Unit-Tests für Code, der in die Produktion geht, abzubilden, kannst du folgende Schritte und GitLab-Features nutzen:

* **GitLab CI/CD Pipelines**:
  + **Entwicklungs- und Testumgebungen**: Richte eine CI/CD-Pipeline ein, die beim Pushen von Code in den develop- oder test-Branch automatisch Unit-Tests ausführt und den Code in der Testumgebung bereitstellt.
  + **Produktionsumgebung**: Vor dem Merge in den main-Branch sollten Unit-Tests in der Pipeline ausgeführt werden, um sicherzustellen, dass der Code produktionsbereit ist.
* **Merge Requests (MRs)**:
  + Nutze Merge Requests, um Änderungen aus dem develop Branch in den test Branch und später in den main Branch zu integrieren. Dabei können in GitLab automatische Tests und Code-Reviews durchgeführt werden.
* **GitLab Auto DevOps**:
  + GitLab bietet eine Auto DevOps-Funktion, die eine Standard-Pipeline mit Build, Test und Deployment für deine Umgebungen automatisiert [[1](https://about.gitlab.com/de-de/topics/version-control/how-implement-version-control/)].