Metodologia — Coleta, Conversão e Incorporação de Funções

1 — Objetivo

Transformar trechos de código úteis (documentação do seaborn e pandas; exemplos do livro Projetos de Ciência de Dados com Python; códigos de projetos pessoais) em **funções Python reutilizáveis** e tipadas, testar sua funcionalidade em **2 datasets** e, quando aprovadas, **incorporar** como entradas na base de conhecimento (`kb.jsonl`), mantendo rastreabilidade e controle mínimo de qualidade.

2 — Regras de implementação obrigatórias (normas)

1. **Assinaturas tipadas**: todos os parâmetros e o retorno devem ter *type hints* (ex.: `df: pd.DataFrame`, `cols: Optional[List[str]] = None`) e o tipo de retorno também (`-> pd.DataFrame` ou outro tipo adequado).
2. **Docstring obrigatória — Google style**: usar `Args:` e `Returns:` Exemplo padrão abaixo.
3. **Nome**: `snake_case`, preferir verbo no início (ex.: `drop_rows_all_zero_except`, `encode_one_hot`).
4. **Sem efeitos colaterais por padrão**: não modificar `df` a menos que `inplace: bool = False` exista e seja `True` explicitamente.
5. **Generalizável**: a função **não** deve depender de nomes fixos de colunas embutidos; se precisar, receber os nomes como argumentos (ex.: `cols: List[str]` ou `prefix: str`).
6. **Tratamento mínimo de erros**: validar tipos e lançar `TypeError`/`ValueError` com mensagens claras quando entradas inválidas.
7. **Permissão para unir blocos**: é permitida a fusão de dois (ou mais) trechos em uma única função se isso fizer sentido e manter uma responsabilidade única e clara.

3 — Template obrigatório de docstring (Google style com `Args:`)

Use **exatamente** este padrão como modelo. Sempre documentar comportamento com NaNs, dtypes e `inplace` quando aplicável.

```
def nome_da_funcao(df: pd.DataFrame, cols: Optional[List[str]] = None, inplace: bool = False) -> pd.DataFrame:
    """
    Descrição curta e direta da função (1-2 frases).

    Args:
        df (pd.DataFrame): DataFrame de entrada. Deve conter as colunas 'x' e 'y' (se aplicável).
        cols (List[str], optional): Lista de colunas a serem processadas. Se None, infere colunas numéricas.
        inplace (bool, optional): Se True, altera o DataFrame original. Default: False.

    Returns:
        pd.DataFrame: DataFrame resultante com as transformações aplicadas (descrição do que muda).
    """
```

4 — Schema final da entrada no KB (JSONL)

Cada função incorporada gera **uma linha JSON** com estes campos **obrigatórios**:

- `id_funcao` (string) — identificador gerado (ver seção 6).
- `titulo` (string) — nome da função, ex.: `"drop_rows_all_zero_except"`.
- `categoria` (string) — macro categoria ex.: `"Exploração e Limpeza de Dados"`.
- `subcategoria` (string) — ex.: `"Tratamento de Dados Faltantes"`.
- `descricao` (string) — frase curta para recuperação/exibição (uma linha).
- `codigo_funcao` (string) — texto completo da função (com docstring e type hints).
- `bibliotecas` (array de strings) — ex.: `["pandas", "numpy"]`.
- `versao` (string) — ex.: `"0.1.0"`.

Exemplo minimal (uma linha do JSONL):

```
{"id_funcao":"<id>","titulo":"drop_rows_all_zero_except","categoria":"Exploração e Limpeza de Dados","subcategoria":"Tratamento de Dados
```

```
Faltantes","descricao":"Remove linhas onde todas as colunas, exceto as excluídas, são zero ou NaN.","codigo_funcao":"def drop_rows_all_zero_except(df: pd.DataFrame, exclude: Optional[List[str]] = None, inplace: bool = False) -> pd.DataFrame:\\n                                                \"\"\"...\"\"\"\\n    ...","bibliotecas":["pandas","numpy"],"versao":"0.1.0"}
```


5 — Fluxo operacional (passo-a-passo)

5.1 Extração

- Identificar blocos candidatos em notebook/arquivo e anotar `origem` (arquivo + célula).
- Aplicar filtros binários:
  - É *essencial*? (S/N) — se não, empurrar para "complemento futuro".
  - É *geralizável* sem depender de colunas nomeadas específicas? (S/N)
  - Não é trivial via API nativa (ex.: não reimplementar `df.mean()`)? (S/N)
- Prosseguir caso as condições 1 e 3 sejam satisfeitos.

5.2 Adaptação para função

- Converter o bloco em função com *type hints* e docstring conforme template.
- Garantir **parâmetros configuráveis** em vez de nomes hard-coded (ex.: `cols`, `prefix`, `mapping`).
- Se necessário unir blocos relacionados em função única, faça — mantendo coesão.

5.3 Preparar testes de incorporação

- Criar **um teste de incorporação** para a função que contenha:
  - Caso A: execução sobre **dataset principal** (ex.: dataset do livro) — usar slice ou caso representativo.
  - Caso B: execução sobre **dataset de validação** (ex.: Titanic) — ou outro dataset clássico aplicável.
- Os asserts do teste **devem ser adequados ao tipo de retorno**: por exemplo verificar tipo do retorno, presença de colunas esperadas, comportamento frente a NaNs, ou tipo `matplotlib.figure.Figure` quando for plot.)
- Guardar o script de teste como `tests/test_<titulo>.py`.

5.4 Execução do teste de incorporação

- Executar testes em **virtualenv** (ambiente isolado).
- Se o teste **passa nos 2 datasets**, seguir para incorporação.
- Se o teste **falha**, seguir ciclo de correção (abaixo).

5.5 Ciclo de correção (até 3 tentativas)

- Corrigir o código (tratamento de NaN, casts, limites de tipos, etc.) e reexecutar os testes.
- Repetir até **3 tentativas**.
- Se **após 3 tentativas** ainda falhar:
  - Registrar a função como **rejeitada** em `rejected_functions.jsonl` com campos: `titulo`, `origem` (arquivo+celula), `motivo_rejeicao` (texto curto), `log_stacktrace`, `tentativas`.
  - Funções rejeitadas ficam no backlog de correção para reavaliação manual futura.

5.6 Incorporação final

- Ao passar nos 2 testes: gerar `id_funcao` (ver seção 6), definir `versao = "0.1.0"`, construir o JSON com os campos obrigatórios e **adicionar uma linha** em `kb.jsonl`.
- Salvar artefatos relacionados localmente:
  - módulo `.py` (com a função e imports),
  - `tests/test_<titulo>.py`,
  - log de execução do teste (texto),
  - `kb.jsonl` atualizado.

6 — Identificador `id_funcao` (unicidade determinística)

Para evitar colisões e gerar ids previsíveis mesmo distribuindo as incorporações ao longo do tempo, **recomendo usar UUID5 determinístico** — baseado em um *namespace fixo* do projeto + uma string identificadora (ex.: `categoria.titulo`).

Exemplo em Python:

```
import uuid

NAMESPACE = uuid.UUID("12345678-1234-5678-1234-567812345678")  # definir uma vez no projeto
```

```python
def make_id_funcao(categoria: str, titulo: str) -> str:
    key = f"{categoria}.{titulo}"
    return str(uuid.uuid5(NAMESPACE, key))

# uso:
id_funcao = make_id_funcao("Exploração e Limpeza de Dados",
"drop_rows_all_zero_except")
```

Vantagem: mesmo que você gere a função amanhã ou daqui a um mês, o `id_funcao` será sempre o mesmo se `categoria` e `titulo` forem iguais — evita duplicação e facilita merge/manual dedup.

Se preferir um identificador legível, alternativa é usar um *slug* determinístico: `"exploracao_drop_rows_all_zero_except_v0_1"`. Escolha uma das duas abordagens; recomendo UUID5 pela robustez.

7 — Critério de aceitação (checklist mínimo para anexar ao KB)

A função só é adicionada ao `kb.jsonl` se **todos** os itens abaixo forem verdadeiros:

- Assinatura com *type hints* completa (parâmetros + retorno).
- Docstring no formato Google style com `Args:` e `Returns:` documentando parâmetros e saída.
- Não depende de nomes de colunas hard-coded (ou esses são parâmetros).
- Teste de incorporação criado e executado; passou nos **2 datasets**.
- Arquivo `kb.jsonl` atualizado com a linha respectiva (`versao` = `"0.1.0"`).

Se algum item falhar, a função entra no ciclo de correção; se após 3 tentativas falhar, vai para `rejected_functions.jsonl`.

8 — Artefatos gerados por função (estrutura local)

Para cada função incorporada, salve localmente:

- `modules/<categoria_slug>/module.py` — contém a função e imports.

- `tests/test_<titulo>.py` — teste de incorporação.
- `logs/<titulo>-<timestamp>.log` — saída dos testes.
- `kb.jsonl` — linha nova adicionada.
- `rejected_functions.jsonl` (apenas para funções rejeitadas).