XISIC.py

```
"""
XI Simpósio Internacional de Climatologia
VII Seminário Internacional de Meteorologia e Climatologia do Amazonas

Sistema de Conversão e Processamento de Dados Meteorológicos INMET.

Este módulo fornece funcionalidades completas para extrair, processar e converter
dados históricos meteorológicos do Instituto Nacional de Meteorologia
(INMET-https://portal.inmet.gov.br/dadoshistoricos) do formato CSV para NetCDF
utilizando a biblioteca xarray, facilitando análises científicas e operacionais
em climatologia e meteorologia.

Desenvolvido especificamente para o minicurso "Processamento de Dados Meteorológicos
com Python" apresentado no XI Simpósio Internacional de Climatologia (XI SIC 2025)
em Belém-PA, promovendo o intercâmbio de conhecimentos em ciência meteorológica
e preparação para discussões da COP30.

Authors:
    Elivaldo Carvalho Rocha
    - Mestrando em Gestão de Risco e Desastres Naturais na Amazônia (PPGGRD-IG-UFPA)
    - Bacharel em Meteorologia (FAMET-IG-UFPA) com especialização em Agrometeorologia
      e climatologia (FAMEESP)
    - MBA em Geotecnologias, Ciência de Dados Geográficos
    - Pós-graduando em Georreferenciamento, Geoprocessamento e Sensoriamento Remoto
    - Graduando em Análise e Desenvolvimento de Sistemas (Faculdade Estácio)
    - Desenvolvedor Full Stack Python

    Prof. Dr. João de Athaydes Silva Júnior
    - Professor Adjunto da UFPA (FAMET e PPGGRD)
    - Coordenador do PPGGRD (2021-2025)
    - Doutor em Desenvolvimento Sustentável do Trópico Úmido (UFPA, 2012)
    - Mestre em Meteorologia (UFCG, 2008)
    - Diretor Administrativo da SBMET (2023-2025)
    - Líder do grupo de pesquisa "Clima na Amazônia"
    - Especialista em Climatologia Aplicada e eventos extremos

Event Context:
    XI Simpósio Internacional de Climatologia & VII Seminário Internacional de
    Meteorologia e Climatologia do Amazonas (XI SIC & VII SIMCA)

    Data: 18-22 de Agosto de 2025
    Local: Auditório SUDAM, Belém-PA
    Organização: SBMET, UFPA, UFAM
    Website: https://sic2025.com.br/

    O XI SIC é um evento bienal que reúne a comunidade científica nacional e
    internacional para discussões sobre clima, sustentabilidade e resiliência,
    servindo como preparação para as temáticas da COP30 em Belém.

Features:
    - Extração automática de arquivos ZIP contendo dados meteorológicos
```

```
    - Processamento robusto de CSVs do INMET com diferentes encodings
    - Conversão para formato NetCDF com estrutura padronizada CF-1.8
    - Filtragem por região, UF e código WMO
    - Tratamento de dados faltantes e inconsistências
    - Preservação de metadados e atributos geográficos
    - Suporte a processamento em lote com controle de progresso
    - Estrutura de dados otimizada para análises com xarray

Dependencies:
    - pandas: Manipulação e análise de dados tabulares
    - xarray: Estruturas de dados multidimensionais para ciências
    - numpy: Computação numérica fundamental
    - pydantic: Validação de dados e serialização
    - pathlib: Manipulação moderna de caminhos de arquivo

Usage Example:
    >>> from XISIC import extract_and_save_csvs, convert_csvs_to_netcdf
    >>>
    >>> # Extrair dados do ZIP
    >>> result = extract_and_save_csvs("/path/to/2024.zip", 2024)
    >>>
    >>> # Converter CSVs para NetCDF
    >>> conversion = convert_csvs_to_netcdf(
    ...     csv_folder="/content/INMET_2024/CSV",
    ...     region="CO",  # Centro-Oeste
    ...     uf="DF",      # Distrito Federal
    ...     all_files=False
    ... )
    >>>
    >>> print(f"Convertidos: {conversion.converted_files} arquivos")

License:
    Desenvolvido para fins educacionais e científicos no contexto do XI SIC 2025.
    Uso livre para pesquisa acadêmica e aplicações meteorológicas.

Version: 1.0.0
Created: Agosto 2025
"""

import glob
import os
import re
import zipfile
from datetime import datetime
from pathlib import Path
from typing import List, Optional, Union

import numpy as np
import pandas as pd
import xarray as xr
from pydantic import BaseModel


class CSVExtractionResult(BaseModel):
```

```python
    """Resultado da operação de extração de arquivos CSV de um ZIP."""

    folder_name: str
    success: bool
    message: str = ""
    file_count: int = 0


class NetCDFConversionResult(BaseModel):
    """Resultado da operação de conversão de CSV para NetCDF."""

    folder_path: str
    success: bool
    message: str = ""
    converted_files: int = 0
    skipped_files: int = 0
    total_files_found: int = 0
    saved_paths: List[str] = []
    failed_files: List[str] = []


def extract_and_save_csvs(zip_path: str, year: int) -> CSVExtractionResult:
    """
    Extrai arquivos de um arquivo zip e os salva em uma pasta nomeada conforme o ano.

    Esta função recebe o caminho de um arquivo zip contendo arquivos CSV e extrai
    todos os arquivos para uma pasta específica formatada como "INMET_{year}",
    onde {year} é o ano fornecido como parâmetro.

    Args:
        zip_path (str): Caminho completo para o arquivo zip que será extraído.
        year (int): Ano que será usado no nome da pasta de destino.

    Returns:
        CSVExtractionResult: Objeto contendo informações sobre a operação de
                             extração, incluindo o nome da pasta criada, sucesso
                             da operação e uma mensagem descritiva.

    Example:
        >>> result = extract_and_save_csvs("/content/2024.zip", 2024)
        >>> print(result.success)
        True
        >>> print(result.folder_name)
        "INMET_2024/CSV"

    Raises:
        FileNotFoundError: Se o arquivo zip não for encontrado.
        Exception: Para outros erros durante a extração do arquivo.

    Note:
        A função cria automaticamente a pasta de destino se ela não existir.
        O arquivo zip deve conter arquivos CSV que serão extraídos para a pasta.
    """
    # Define o nome da pasta
```

```python
    folder_name = f"INMET_{year}/CSV"

    # Cria a pasta se ela não existir
    if not os.path.exists(folder_name):
        os.makedirs(folder_name)

    # Extrai o arquivo zip
    with zipfile.ZipFile(zip_path, 'r') as zip_ref:
        zip_ref.extractall(folder_name)

    # Conta quantos arquivos foram extraídos
    file_count = len(os.listdir(folder_name))

    print(f"Files extracted to {folder_name} - Total: {file_count} files")

    # Retorna o resultado
    return CSVExtractionResult(
        folder_name=folder_name,
        success=True,
        message=f"Files extracted successfully. Total files: {file_count}",
        file_count=file_count
    )


def debug_inmet_file(csv_file_path: str) -> tuple:
    """
    Função para debugar e entender a estrutura do arquivo INMET.

    Args:
        csv_file_path (str): Caminho para o arquivo CSV do INMET.

    Returns:
        tuple: Tupla contendo (encoding, metadata, columns).

    Raises:
        ValueError: Se não for possível ler o arquivo com nenhum encoding.
    """
    print("=== DEBUG DO ARQUIVO INMET ===")

    # Tentar diferentes encodings
    encoding = None
    lines = []

    for enc in ['latin1', 'utf-8']:
        try:
            with open(csv_file_path, 'r', encoding=enc) as f:
                lines = f.readlines()
            print(f"✅ Arquivo lido com encoding: {enc}")
            encoding = enc
            break
        except UnicodeDecodeError:
            continue
    else:
        raise ValueError("Não foi possível ler o arquivo com nenhum encoding")
```

```
214
215     print(f"Total de linhas: {len(lines)}")
216
217     # Mostrar primeiras 10 linhas
218     print("\n=== PRIMEIRAS 10 LINHAS ===")
219     for i, line in enumerate(lines[:10]):
220         print(f"Linha {i+1}: {repr(line.strip())}")
221
222     # Analisar metadados
223     print("\n=== METADADOS ===")
224     metadata = {}
225     for i in range(8):
226         parts = lines[i].strip().split(';')
227         if len(parts) >= 2:
228             key, value = parts[0], parts[1]
229             metadata[key.replace(':', '')] = value
230             print(f"{key}: {value}")
231
232     # Analisar cabeçalho
233     print(f"\n=== CABEÇALHO (linha 9) ===")
234     header_line = lines[8].strip()
235     columns = header_line.split(';')
236     print(f"Total de colunas: {len(columns)}")
237     for i, col in enumerate(columns):
238         print(f"  {i+1}: '{col}'")
239
240     # Analisar primeira linha de dados
241     print(f"\n=== PRIMEIRA LINHA DE DADOS (linha 10) ===")
242     if len(lines) > 9:
243         data_line = lines[9].strip()
244         data_values = data_line.split(';')
245         print(f"Valores: {data_values}")
246         print(f"Total de valores: {len(data_values)}")
247
248     return encoding, metadata, columns
249
250
251 def parse_inmet_csv_to_netcdf_robust(csv_file_path: str,
252                                      output_netcdf_path: Optional[str] = None,
253                                      debug: bool = True) -> xr.Dataset:
254     """
255     Versão robusta para converter arquivo CSV do INMET para NetCDF.
256
257     Args:
258         csv_file_path (str): Caminho para o arquivo CSV do INMET.
259         output_netcdf_path (str, optional): Caminho de saída do arquivo NetCDF.
260         debug (bool, optional): Se True, mostra informações de debug.
261
262     Returns:
263         xr.Dataset: Dataset xarray estruturado.
264
265     Raises:
266         ValueError: Se não for possível encontrar colunas essenciais.
267         Exception: Para outros erros durante o processamento.
```

```
    """
    if debug:
        encoding, metadata, columns = debug_inmet_file(csv_file_path)
    else:
        # Ler arquivo silenciosamente
        encoding = None
        lines = []

        for enc in ['cp1252', 'latin1', 'utf-8']:
            try:
                with open(csv_file_path, 'r', encoding=enc) as f:
                    lines = f.readlines()
                encoding = enc
                break
            except UnicodeDecodeError:
                continue

        # Extrair metadados
        metadata = {}
        for i in range(8):
            parts = lines[i].strip().split(';')
            if len(parts) >= 2:
                key, value = parts[0], parts[1]
                metadata[key.replace(':', '')] = value

    # Extrair informações do nome do arquivo
    filename = Path(csv_file_path).stem
    file_parts = filename.split('_')

    if len(file_parts) >= 5:
        region = file_parts[1]
        uf = file_parts[2]
        wmo_code = file_parts[3]
        station_name = file_parts[4]
    else:
        # Fallback se o nome do arquivo não seguir o padrão
        region = metadata.get('REGIAO', 'Unknown')
        uf = metadata.get('UF', 'Unknown')
        wmo_code = metadata.get('CODIGO (WMO)', 'Unknown')
        station_name = metadata.get('ESTACAO', 'Unknown')

    # Converter coordenadas
    latitude = float(metadata['LATITUDE'].replace(',', '.'))
    longitude = float(metadata['LONGITUDE'].replace(',', '.'))
    altitude = float(metadata['ALTITUDE'].replace(',', '.'))

    if debug:
        print(f"\n=== INFORMAÇÕES EXTRAÍDAS ===")
        print(f"Região: {region}")
        print(f"UF: {uf}")
        print(f"WMO: {wmo_code}")
        print(f"Estação: {station_name}")
        print(f"Lat: {latitude}, Lon: {longitude}, Alt: {altitude}")

```

```
    # Ler dados com pandas
    try:
        df = pd.read_csv(
            csv_file_path,
            sep=';',
            decimal=',',
            skiprows=8,
            header=0,
            encoding=encoding,
            na_values=['', ' ', 'NULL', 'null', '-9999'],
            keep_default_na=True
        )

        if debug:
            print(f"\n=== DATAFRAME CARREGADO ===")
            print(f"Shape: {df.shape}")
            print(f"Colunas: {list(df.columns)}")
            print(f"Primeiras 3 linhas:")
            print(df.head(3))

    except Exception as e:
        print(f"Erro ao ler CSV com pandas: {e}")
        raise

    # Limpar colunas vazias
    df = df.dropna(axis=1, how='all')

    # Renomear colunas principais
    column_renames = {}
    for col in df.columns:
        if 'Data' in col:
            column_renames[col] = 'date'
        elif 'Hora' in col and 'UTC' in col:
            column_renames[col] = 'hour_utc'

    df = df.rename(columns=column_renames)

    if debug:
        print(f"\nColunas após renomeação: {list(df.columns)}")

    # Verificar se temos as colunas essenciais
    if 'date' not in df.columns or 'hour_utc' not in df.columns:
        raise ValueError("Não foi possível encontrar colunas de data e hora")

    # Processar datetime
    try:
        # Limpar e processar campo de hora
        hour_clean = (df['hour_utc']
                      .astype(str)
                      .str.replace(' UTC', '')
                      .str.replace('UTC', '')
                      .str.strip())
        df['hour_clean'] = hour_clean

```

```
        # Criar datetime
        df['datetime'] = pd.to_datetime(
            df['date'].astype(str) + ' ' + df['hour_clean'],
            format='%Y/%m/%d %H%M',
            errors='coerce'
        )

        # Remover linhas com datetime inválido
        df = df.dropna(subset=['datetime'])

        if debug:
            print(f"Linhas válidas após processamento de datetime: {len(df)}")

    except Exception as e:
        print(f"Erro ao processar datetime: {e}")
        raise

    # Extrair componentes de data e hora
    df['date_only'] = df['datetime'].dt.date
    df['hour_only'] = df['datetime'].dt.hour

    # Criar coordenadas
    dates = pd.to_datetime(sorted(df['date_only'].unique()))
    hours = sorted(df['hour_only'].unique())

    if debug:
        print(f"\nPeríodo: {dates[0]} a {dates[-1]}")
        print(f"Horas disponíveis: {hours}")

    # Identificar variáveis meteorológicas
    excluded_cols = [
        'date', 'hour_utc', 'hour_clean', 'datetime', 'date_only', 'hour_only'
    ]
    met_variables = [col for col in df.columns if col not in excluded_cols]

    if debug:
        print(f"Variáveis meteorológicas: {met_variables}")

    # Criar dataset xarray
    data_vars = {}

    for var in met_variables:
        # Criar matriz 2D para dados
        var_matrix = np.full((len(dates), len(hours)), np.nan)

        for i, date in enumerate(dates):
            for j, hour in enumerate(hours):
                mask = ((df['date_only'] == date.date()) &
                        (df['hour_only'] == hour))
                if mask.any():
                    value = df.loc[mask, var].iloc[0]
                    if pd.notna(value) and value != '':
                        try:
                            var_matrix[i, j] = float(value)
```

```
                        except (ValueError, TypeError):
                            pass  # Manter como NaN

        # Reshape para incluir todas as dimensões
        var_array = var_matrix.reshape(1, 1, 1, len(dates), len(hours), 1, 1, 1)

        data_vars[var] = (
            ['region', 'uf', 'wmo_code', 'date', 'hour_utc', 'lat', 'lon', 'alt'],
            var_array
        )

    # Coordenadas
    coords = {
        'region': [region],
        'uf': [uf],
        'wmo_code': [wmo_code],
        'date': dates,
        'hour_utc': hours,
        'lat': [latitude],
        'lon': [longitude],
        'alt': [altitude]
    }

    # Criar dataset
    ds = xr.Dataset(data_vars=data_vars, coords=coords)

    # Adicionar atributos globais
    ds.attrs.update({
        'source': 'INMET',
        'station': station_name,
        'date_of_foundation': metadata.get('DATA DE FUNDACAO', 'Unknown'),
        'title': f'Dados meteorológicos horários - {station_name}',
        'institution': 'Instituto Nacional de Meteorologia (INMET)',
        'region': region,
        'uf': uf,
        'wmo_code': wmo_code,
        'latitude': latitude,
        'longitude': longitude,
        'altitude_m': altitude,
        'creation_date': datetime.now().strftime('%Y-%m-%d %H:%M:%S'),
        'conventions': 'CF-1.8'
    })

    # Adicionar atributos às coordenadas
    ds['lat'].attrs = {'units': 'degrees_north', 'long_name': 'Latitude'}
    ds['lon'].attrs = {'units': 'degrees_east', 'long_name': 'Longitude'}
    ds['alt'].attrs = {'units': 'm', 'long_name': 'Altitude above sea level'}
    ds['date'].attrs = {'long_name': 'Date'}
    ds['hour_utc'].attrs = {'units': 'hours', 'long_name': 'Hour in UTC'}

    # Salvar se especificado
    if output_netcdf_path:
        ds.to_netcdf(output_netcdf_path)
        if debug:
```

```
            print(f"\n✅ Arquivo NetCDF salvo em: {output_netcdf_path}")

    return ds


def _normalize_filter(filter_value: Optional[Union[str, List[str]]]) -> Optional[List[str]]:
    """
    Função auxiliar para normalizar filtros.

    Args:
        filter_value: Valor do filtro (string ou lista de strings).

    Returns:
        Lista de strings em maiúsculo ou None.
    """
    if filter_value is None:
        return None
    if isinstance(filter_value, str):
        return [filter_value.upper()]
    return [f.upper() for f in filter_value]


def convert_csvs_to_netcdf(
    csv_folder: Union[str, Path],
    region: Optional[Union[str, List[str]]] = None,
    uf: Optional[Union[str, List[str]]] = None,
    wmo_code: Optional[Union[str, List[str]]] = None,
    all_files: bool = False,
    debug: bool = False,
    skip_existing: bool = True
) -> NetCDFConversionResult:
    """
    Converte CSVs de uma pasta para NetCDF com opções de filtragem.

    Salva arquivos em uma subpasta 'NETCDF'.

    Padrão de entrada: INMET_{region}_{uf}_{wmo_code}_{station}_{start_date}_A_{end_date}.CSV
    Padrão de saída: INMET_{region}_{uf}_{wmo_code}_{station}_{start_date}_A_{end_date}.nc
    Pasta de destino: /content/INMET_2024/NETCDF/

    Args:
        csv_folder: Pasta contendo os CSVs (ex: '/content/INMET_2024/CSV').
        region: Filtro por região(s) (N, NE, CO, SE, S).
        uf: Filtro por UF(s) (siglas de 2 letras).
            AC, AL, AP, AM, BA, CE, DF, ES, GO, MA, MT, MS, MG, PA, PB, PR, PE,
            PI, RJ, RN, RS, RO, RR, SC, SP, SE, TO.
        wmo_code: Filtro por código(s) WMO.
        all_files: Se True, converte todos os arquivos ignorando filtros.
        debug: Se True, mostra informações detalhadas de debug.
        skip_existing: Se True (padrão), pula arquivos NetCDF que já existem.

    Returns:
```

```
        NetCDFConversionResult: Resultado da conversão com caminhos dos
                                arquivos salvos.

    Example:
        # Converter todos os arquivos
        result = convert_csvs_to_netcdf('/content/INMET_2024/CSV', all_files=True)

        # Converter apenas Brasília
        result = convert_csvs_to_netcdf('/content/INMET_2024/CSV', uf='DF')

        # Converter região Centro-Oeste
        result = convert_csvs_to_netcdf('/content/INMET_2024/CSV', region='CO')

        # Múltiplos filtros
        result = convert_csvs_to_netcdf('/content/INMET_2024/CSV',
                                        region=['CO', 'SE'],
                                        uf=['DF', 'SP'])
    """
    # Converter para Path e validar
    csv_path = Path(csv_folder)
    if not csv_path.exists():
        return NetCDFConversionResult(
            folder_path=str(csv_path),
            success=False,
            message=f"Pasta {csv_path} não encontrada"
        )

    # Obter pasta pai e criar pasta de destino
    parent_folder = csv_path.parent
    netcdf_folder = parent_folder / "NETCDF"
    netcdf_folder.mkdir(exist_ok=True)

    print(f"🔍 Procurando arquivos CSV em: {csv_path}")
    print(f"💾 Salvando NetCDF em: {netcdf_folder}")

    # Encontrar todos os arquivos CSV do INMET
    pattern = csv_path / "INMET_*.CSV"
    all_csv_files = list(glob.glob(str(pattern)))

    if not all_csv_files:
        return NetCDFConversionResult(
            folder_path=str(csv_path),
            success=False,
            message="Nenhum arquivo CSV do INMET encontrado"
        )

    print(f"📁 Encontrados {len(all_csv_files)} arquivos CSV do INMET")

    # Normalizar filtros
    region_filter = _normalize_filter(region)
    uf_filter = _normalize_filter(uf)
    wmo_filter = _normalize_filter(wmo_code)

    # Filtrar arquivos se necessário
```

```
    if all_files:
        filtered_files = all_csv_files
        print("🌍 Modo ALL_FILES ativo - processando todos os arquivos")
    else:
        filtered_files = []

        for file_path in all_csv_files:
            filename = Path(file_path).name

            # Extrair componentes do nome do arquivo
            # Padrão: INMET_CO_DF_A001_BRASILIA_01-01-2024_A_31-12-2024.CSV
            parts = filename.replace('.CSV', '').split('_')

            if len(parts) < 5:
                if debug:
                    print(f"⚠️ Arquivo com padrão inválido: {filename}")
                continue

            file_region = parts[1].upper()
            file_uf = parts[2].upper()
            file_wmo = parts[3].upper()

            # Aplicar filtros
            include_file = True

            if region_filter and file_region not in region_filter:
                include_file = False
            if uf_filter and file_uf not in uf_filter:
                include_file = False
            if wmo_filter and file_wmo not in wmo_filter:
                include_file = False

            if include_file:
                filtered_files.append(file_path)
                if debug:
                    print(f"✅ Incluído: {filename}")
            elif debug:
                print(f"❌ Filtrado: {filename}")

    if not filtered_files:
        filters_applied = []
        if region_filter:
            filters_applied.append(f"region={region_filter}")
        if uf_filter:
            filters_applied.append(f"uf={uf_filter}")
        if wmo_filter:
            filters_applied.append(f"wmo_code={wmo_filter}")

        filter_text = ", ".join(filters_applied) if filters_applied else "nenhum"

        return NetCDFConversionResult(
            folder_path=str(csv_path),
            success=False,
            total_files_found=len(all_csv_files),
```

```
            message=f"Nenhum arquivo encontrado com os filtros: {filter_text}"
        )

    print(f"🎯 Arquivos selecionados para conversão: {len(filtered_files)}")

    # Variáveis para rastrear o progresso
    saved_paths = []
    failed_files = []
    skipped_files = []
    converted_files = 0

    # Converter cada arquivo
    for i, csv_file_path in enumerate(filtered_files):
        try:
            filename = Path(csv_file_path).name
            print(f"\n📊 [{i+1}/{len(filtered_files)}] Processando: {filename}")

            # Definir caminho de saída
            netcdf_filename = filename.replace('.CSV', '.nc')
            output_path = netcdf_folder / netcdf_filename

            # Verificar se já existe e deve pular
            if output_path.exists() and skip_existing:
                print(f"⏭️ Pulando arquivo existente: {netcdf_filename}")
                skipped_files.append(csv_file_path)
                saved_paths.append(str(output_path))

                # Mostrar informações do arquivo existente
                file_size_mb = output_path.stat().st_size / (1024 * 1024)
                print(f"   💾 Tamanho existente: {file_size_mb:.1f} MB")
                continue

            # Verificar se já existe mas deve sobrescrever
            if output_path.exists() and not skip_existing:
                print(f"⚠️ Sobrescrevendo arquivo existente: {netcdf_filename}")

            # Converter usando a função existente
            dataset = parse_inmet_csv_to_netcdf_robust(
                csv_file_path=csv_file_path,
                output_netcdf_path=str(output_path),
                debug=False  # Desabilitar debug individual
            )

            # Verificar se a conversão foi bem-sucedida
            if dataset is not None and output_path.exists():
                saved_paths.append(str(output_path))
                converted_files += 1

                # Mostrar informações básicas do dataset
                station_name = dataset.attrs.get('station', 'Unknown')
                total_records = len(dataset.date) * len(dataset.hour_utc)
                file_size_mb = output_path.stat().st_size / (1024 * 1024)

                print(f"✅ Sucesso: {station_name}")
```

```
                print(f"    📊 Registros: {total_records}")
                print(f"    💾 Tamanho: {file_size_mb:.1f} MB")
                print(f"    💿 Salvo: {netcdf_filename}")
            else:
                failed_files.append(csv_file_path)
                print(f"❌ Falha na conversão de: {filename}")

        except Exception as e:
            failed_files.append(csv_file_path)
            print(f"❌ Erro ao processar {filename}: {str(e)}")
            if debug:
                import traceback
                traceback.print_exc()

    # Compilar resultado final
    total_processed = converted_files + len(skipped_files)
    success = total_processed > 0

    if success:
        message = f"Processamento concluído: {converted_files} convertidos"
        if skipped_files:
            message += f", {len(skipped_files)} pulados (já existiam)"
        if failed_files:
            message += f", {len(failed_files)} falharam"
    else:
        message = f"Nenhum arquivo foi processado. {len(failed_files)} falharam"

    # Mostrar resumo final
    _print_conversion_summary(
        csv_path, netcdf_folder, all_csv_files, filtered_files,
        converted_files, skipped_files, failed_files, skip_existing,
        saved_paths, debug
    )

    return NetCDFConversionResult(
        folder_path=str(netcdf_folder),
        success=success,
        message=message,
        converted_files=converted_files,
        skipped_files=len(skipped_files),
        total_files_found=len(all_csv_files),
        saved_paths=saved_paths,
        failed_files=[str(Path(f).name) for f in failed_files]
    )


def _print_conversion_summary(csv_path: Path,
                              netcdf_folder: Path,
                              all_csv_files: List[str],
                              filtered_files: List[str],
                              converted_files: int,
                              skipped_files: List[str],
                              failed_files: List[str],
                              skip_existing: bool,
```

```
                            saved_paths: List[str],
                            debug: bool) -> None:
    """
    Imprime o resumo da conversão.

    Args:
        csv_path: Caminho da pasta CSV.
        netcdf_folder: Caminho da pasta NetCDF.
        all_csv_files: Lista de todos os arquivos CSV encontrados.
        filtered_files: Lista de arquivos filtrados.
        converted_files: Número de arquivos convertidos.
        skipped_files: Lista de arquivos pulados.
        failed_files: Lista de arquivos que falharam.
        skip_existing: Se deve pular arquivos existentes.
        saved_paths: Lista de caminhos salvos.
        debug: Se deve mostrar informações de debug.
    """
    print(f"\n🎯 ===== RESUMO DA CONVERSÃO =====")
    print(f"📁 Pasta origem: {csv_path}")
    print(f"💾 Pasta destino: {netcdf_folder}")
    print(f"🔍 Arquivos encontrados: {len(all_csv_files)}")
    print(f"🎯 Arquivos selecionados: {len(filtered_files)}")
    print(f"✅ Novos arquivos convertidos: {converted_files}")
    print(f"⏭️ Arquivos pulados (já existiam): {len(skipped_files)}")
    print(f"❌ Conversões falharam: {len(failed_files)}")
    print(f"📊 Total processado: {converted_files + len(skipped_files)}"
          f"/{len(filtered_files)}")

    mode_text = "pular arquivos existentes" if skip_existing else "sobrescrever arquivos existentes"
    print(f"ℹ️ Modo skip_existing={skip_existing} ({mode_text})")

    if failed_files:
        print(f"\n❌ Arquivos que falharam:")
        for failed_file in failed_files[:5]:
            print(f"   • {Path(failed_file).name}")
        if len(failed_files) > 5:
            print(f"   ... e mais {len(failed_files) - 5} arquivos")

    if skipped_files and debug:
        print(f"\n⏭️ Arquivos pulados (amostra):")
        for skipped_file in skipped_files[:3]:
            print(f"   • {Path(skipped_file).name}")
        if len(skipped_files) > 3:
            print(f"   ... e mais {len(skipped_files) - 3} arquivos")

    if saved_paths:
        total_size_mb = sum(Path(p).stat().st_size for p in saved_paths) / (1024 * 1024)
        print(f"\n💾 Tamanho total dos NetCDF: {total_size_mb:.1f} MB")
        print(f"📊 Média por arquivo: {total_size_mb/len(saved_paths):.1f} MB")
        if converted_files > 0:
            new_files_size = (sum(Path(p).stat().st_size
                                  for p in saved_paths[-converted_files:])
                              / (1024 * 1024))
```

```
805             print(f"🆕 Tamanho dos novos arquivos: {new_files_size:.1f} MB")
806
```