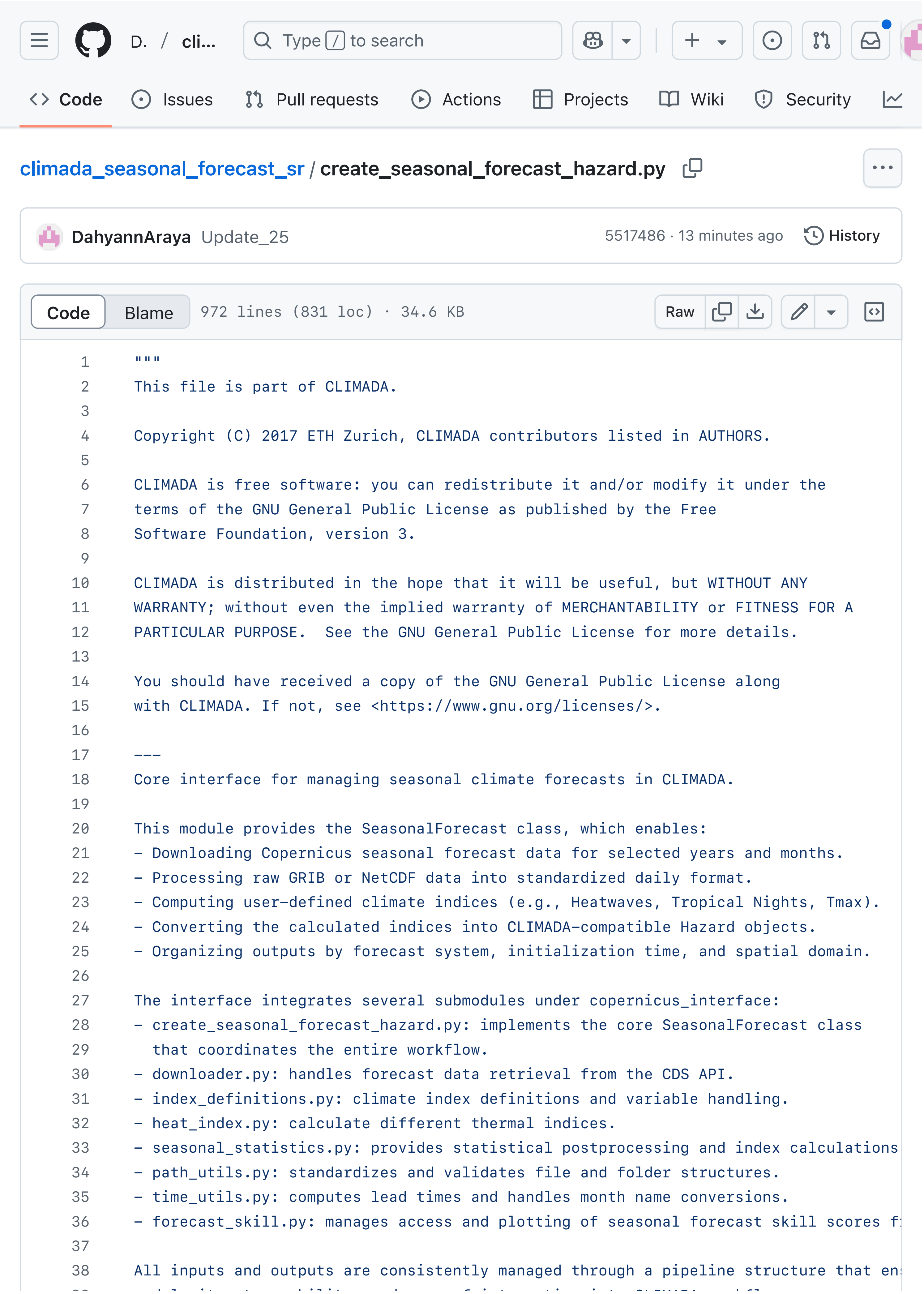

climada_seasonal_forecast_sr / create_seasonal_forecast_hazard.py

DahyannAraya Update_25 5517486 · 13 minutes ago

972 lines (831 loc) · 34.6 KB

```
"""
This file is part of CLIMADA.

Copyright (C) 2017 ETH Zurich, CLIMADA contributors listed in AUTHORS.

CLIMADA is free software: you can redistribute it and/or modify it under the
terms of the GNU General Public License as published by the Free
Software Foundation, version 3.

CLIMADA is distributed in the hope that it will be useful, but WITHOUT ANY
WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.  See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along
with CLIMADA. If not, see <https://www.gnu.org/licenses/>.

---
Core interface for managing seasonal climate forecasts in CLIMADA.

This module provides the SeasonalForecast class, which enables:
- Downloading Copernicus seasonal forecast data for selected years and months.
- Processing raw GRIB or NetCDF data into standardized daily format.
- Computing user-defined climate indices (e.g., Heatwaves, Tropical Nights, Tmax).
- Converting the calculated indices into CLIMADA-compatible Hazard objects.
- Organizing outputs by forecast system, initialization time, and spatial domain.

The interface integrates several submodules under copernicus_interface:
- create_seasonal_forecast_hazard.py: implements the core SeasonalForecast class
  that coordinates the entire workflow.
- downloader.py: handles forecast data retrieval from the CDS API.
- index_definitions.py: climate index definitions and variable handling.
- heat_index.py: calculate different thermal indices.
- seasonal_statistics.py: provides statistical postprocessing and index calculations
- path_utils.py: standardizes and validates file and folder structures.
- time_utils.py: computes lead times and handles month name conversions.
- forecast_skill.py: manages access and plotting of seasonal forecast skill scores f

All inputs and outputs are consistently managed through a pipeline structure that en
```

```python
modularity, traceability, and ease of integration into CLIMADA workflows.

"""
import calendar
import logging
from datetime import date
from pathlib import Path, PosixPath
from typing import List

import cartopy.crs as ccrs
import cartopy.feature as cfeature
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import xarray as xr

from climada import CONFIG
from climada.hazard import Hazard
from climada_petals.hazard.copernicus_interface.downloader import download_data
from climada_petals.hazard.copernicus_interface.index_definitions import (
    IndexSpecEnum,
    get_short_name_from_variable,
)
import climada_petals.hazard.copernicus_interface.seasonal_statistics as seasonal_sta
from climada_petals.hazard.copernicus_interface.time_utils import (
    month_name_to_number,
    calculate_leadtimes,
)
from climada_petals.hazard.copernicus_interface.path_utils import (
check_existing_files, get_file_path
)


# set path to store data
DATA_OUT = CONFIG.hazard.copernicus.seasonal_forecasts.dir()
LOGGER = logging.getLogger(__name__)


# ----- Main Class -----
class SeasonalForecast:
    """
    Class for managing the download, processing, and analysis of seasonal climate fo
    """

    def __init__(
        self,
        index_metric,
        year_list,
        forecast_period,
        initiation_month,
        bounds,
        data_format,
        originating_centre,
        system,
        data_out=None,
    ):
        """
        Initialize the SeasonalForecast instance with user-defined parameters for in

        Parameters
        ----------
        index_metric : str
            Climate index to calculate (e.g., "HW", "TR", "Tmax").
        year_list : list of int
            List of years for which data should be downloaded and processed.
        lead_time_months : list of str or int
            List specifying the start and end month (given as integers or strings)
            of the valid forecast period. Must contain exactly two elements.
        initiation_month : list of str
            List of initiation months for the forecast (e.g., ["March", "April"]).
        bounds : list of float
            Bounding box values in EPSG 4326 format: (min_lon, min_lat, max_lon, max_
        data_format : str
            Format of the downloaded data. Either "grib" or "netcdf".
        originating_centre : str
            Data provider (e.g., "dwd").
        system : str
            Forecast system configuration (e.g., "21").
        data_out : pathlib.Path, optional
            Output directory for storing downloaded and processed data. If None,
            uses a default directory specified in the configuration.

        Raises
        ------
        ValueError
            If the valid period does not contain exactly two months.
        """
        # initiate initiation month, valid period, and leadtimes
        valid_period = forecast_period
        if not isinstance(initiation_month, list):
            initiation_month = [initiation_month]
        if not isinstance(valid_period, list) or len(valid_period) != 2:
            raise ValueError("Valid period must be a list of two months.")
        self.initiation_month_str = [
            f"{month_name_to_number(month):02d}" for month in initiation_month
        ]
        self.valid_period = [month_name_to_number(month) for month in valid_period]
        self.valid_period_str = "_".join(
            [f"{month:02d}" for month in self.valid_period]
        )

        self.index_metric = index_metric
        self.year_list = year_list
        self.bounds = bounds
        self.bounds_str = (
            f"boundsN{int(self.bounds[0])}_S{int(self.bounds[1])}_"
            f"boundsN{int(self.bounds[0])}_S{int(self.bounds[1])}_"
            f"E{int(self.bounds[2])}_W{int(self.bounds[3])}"
            )
        self.data_format = data_format
        self.originating_centre = originating_centre
        self.system = system

        # initialze base directory
        self.data_out = Path(data_out) if data_out else DATA_OUT

        # Get index specifications
        index_spec = IndexSpecEnum.get_info(self.index_metric)
        self.variables = index_spec.variables
        self.variables_short = [
            get_short_name_from_variable(var) for var in self.variables
        ]


    ##########  Index Metadata Utilities  ##########

    def explain_index(self, index_metric=None, print_flag=False):
        """
        Retrieve and display information about a specific climate index.

        This function provides an explanation and the required input variables for
        the selected climate index. If no index is provided, the instance's
        `index_metric` is used.

        Parameters
        ----------
        index_metric : str, optional
            Climate index to explain (e.g., 'HW', 'TR', 'Tmax'). If None, uses the
            instance's index_metric.
        print_flag : bool, optional
            If True, prints the explanation. Default is False.

        Returns
        -------
        str
            Text description of the index explanation and required input variables.

        Notes
        -----
        The index information is retrieved from `IndexSpecEnum.get_info`.
        """
        index_metric = index_metric or self.index_metric
        response = (
            f"Explanation for {index_metric}: "
            f"{IndexSpecEnum.get_info(index_metric).explanation} "
        )
        response += (
            "Required variables: "
            f"{', '.join(IndexSpecEnum.get_info(index_metric).variables)}"
        )
        if print_flag:
            print(response)
        return response


    ##########  Path Utilities  ##########

    def get_pipeline_path(self, year, initiation_month_str, data_type):
        """
        Provide (and possibly create) file paths for forecast pipeline.

        Parameters
        ----------
        year : int
            Year of the forecast initiation.
        init_month : str
            Initiation month as two-digit string (e.g., '03' for March).
        data_type : str
            Type of data to access ('downloaded_data', 'processed_data', 'indices',

        Returns
        -------
        Path or dict of Path
            Path to the requested file(s). For 'indices', returns a dictionary with
            'daily', 'monthly', 'stats'.

        Raises
        ------
        ValueError
            If unknown data_type is provided.

        Notes
        -----
        File structure:
        {base_dir}/{originating_centre}/sys{system}/{year}/init{init_month}/valid{va
        /{data_type}
        """
        file_path = get_file_path(
            self.data_out,
            self.originating_centre,
            year,
            initiation_month_str,
            self.valid_period_str,
            data_type,
            self.index_metric,
            self.bounds_str,
            self.system,
            self.data_format,
        )

        # create directory if not existing
        if data_type == "indices":
```

```python
        if data_type == "indices":
            file_path["monthly"].parent.mkdir(parents=True, exist_ok=True)
        else:
            file_path.parent.mkdir(parents=True, exist_ok=True)

        return file_path

    ##########  Download and process ##########

    def _download(self, overwrite=False):
        """
        Download seasonal forecast data for the specified years and initiation month

        This function downloads the raw forecast data files for each year and initia
        defined in the instance configuration. The data is downloaded in the specifi
        ('grib' or 'netcdf') and stored in the configured directory structure.

        Parameters
        ----------
        overwrite : bool, optional
            If True, existing downloaded files will be overwritten. Default is False

        Returns
        -------
        dict
            Dictionary with keys of the form "<year>_init<month>_valid<valid_period>
            and values corresponding to the downloaded data file paths.

        Notes
        -----
        The data is downloaded using the `_download_data` function and follows the d
        structure defined in `get_pipeline_path`. The bounding box is automatically 
        to CDS (Climate Data Store) format before download.
        """
        output_files = {}
        bounds_cds_order = [
            self.bounds[3],
            *self.bounds[:3],
        ]  # convert bounds to CDS order
        for year in self.year_list:
            for month_str in self.initiation_month_str:
                leadtimes = calculate_leadtimes(year, int(month_str), self.valid_per

                # Generate output file name
                downloaded_data_path = self.get_pipeline_path(
                    year, month_str, "downloaded_data"
                )

                output_files[f"{year}_init{month_str}_valid{self.valid_period_str}"]
                    _download_data(
                        downloaded_data_path,
                        overwrite,
                        self.variables,
                        year,
                        month_str,
                        self.data_format,
                        self.originating_centre,
                        self.system,
                        bounds_cds_order,
                        leadtimes,
                    )
                )

        return output_files

    def _process(self, overwrite=False):
        """
        Process the downloaded forecast data into daily NetCDF format.

        This function processes the raw downloaded data files into a standardized
        daily NetCDF format, applying basic aggregation operations (mean, max, min).
        The processed files are saved in the configured output directory.

        Parameters
        ----------
        overwrite : bool, optional
            If True, existing processed files will be overwritten. Default is False.

        Returns
        -------
        dict
            Dictionary with keys of the form "<year>_init<month>_valid<valid_period>
            and values corresponding to the processed NetCDF file paths.

        Notes
        -----
        The processing applies a daily coarsening operation and aggregates the data.
        The processed data is saved in NetCDF format in the directory defined by
        `get_pipeline_path`. Processing is performed using the `_process_data` funct
        """
        processed_files = {}
        for year in self.year_list:
            for month_str in self.initiation_month_str:
                # Locate input file name
                downloaded_data_path = self.get_pipeline_path(
                    year, month_str, "downloaded_data"
                )
                # Generate output file name
                processed_data_path = self.get_pipeline_path(
                    year, month_str, "processed_data"
                )

                processed_files[
                    f"{year}_init{month_str}_valid{self.valid_period_str}"
                ] = _process_data(
                    processed_data_path,
```

```python
                    processed_data_path,
                    overwrite,
                    downloaded_data_path,
                    self.variables_short,
                    self.data_format,
                )

        return processed_files

    def download_and_process_data(self, overwrite=False):
        """
        Download and process seasonal climate forecast data.

        This function performs the complete data pipeline by first downloading
        the raw forecast data for the specified years and initiation months,
        and then processing the downloaded data into a daily NetCDF format.

        Parameters
        ----------
        overwrite : bool, optional
            If True, existing downloaded and processed files will be overwritten. De

        Returns
        -------
        dict
            Dictionary containing two keys:
            - "downloaded_data": dict with file paths to downloaded raw data.
            - "processed_data": dict with file paths to processed NetCDF data.

        Raises
        ------
        Exception
            If an error occurs during download or processing, such as invalid input
            or file system issues.

        Notes
        -----
        This is a high-level method that internally calls `_download()` and `_proces
        The file structure and naming follow the configuration defined in `get_pipel
        """

        # Call high-level methods for downloading and processing
        created_files = {}
        try:
            # 1) Attempt downloading data
            created_files["downloaded_data"] = self._download(overwrite=overwrite)
            # 2) Attempt processing data
            created_files["processed_data"] = self._process(overwrite=overwrite)
        except Exception as error:
            # Catch reversed valid_period or any other ValueError from calculate_lea
            raise RuntimeError(f"Download/process aborted: {error}") from error

        return created_files
    ##########  Calculate index ##########

    def calculate_index(
        self,
        overwrite=False,
        hw_threshold=27,
        hw_min_duration=3,
        hw_max_gap=0,
        tr_threshold=20,
    ):
        """
        Calculate the specified climate index based on the downloaded forecast data.

        This function processes the downloaded or processed forecast data to compute
        the selected climate index (e.g., Heatwave days, Tropical Nights) according
        to the parameters defined for the index.

        Parameters
        ----------
        overwrite : bool, optional
            If True, existing index files will be overwritten. Default is False.
        hw_threshold : float, optional
            Temperature threshold for heatwave days index calculation. Default is 27
        hw_min_duration : int, optional
            Minimum duration (in days) of consecutive conditions for a heatwave even
        hw_max_gap : int, optional
            Maximum allowable gap (in days) between conditions to still
            consider as a single heatwave event. Default is 0.
        tr_threshold : float, optional
            Temperature threshold for tropical nights index calculation. Default is

        Returns
        -------
        dict
            Dictionary with keys of the form "<year>_init<month>_valid<valid_period>
            and values corresponding to the output NetCDF index files (daily, monthl

        Raises
        ------
        Exception
            If index calculation fails due to missing files or processing errors.

        Notes
        -----
        The input files used depend on the index:
        - For 'TX30', 'TR', and 'HW', the raw downloaded GRIB data is used.
        - For other indices, the processed NetCDF data is used.

        The calculation is performed using the `_calculate_index` function and resul
        are saved in the configured output directory structure.
        """
        index_outputs = {}
```

```python

        # Iterate over each year and initiation month
        for year in self.year_list:
            for month_str in self.initiation_month_str:
                LOGGER.info(
                    "Processing index %s for year %s, initiation month %s.",
                    self.index_metric,
                    year,
                    month_str,
                )

                # Determine the input file based on index type
                if self.index_metric in ["TX30", "TR", "HW"]:  # Metrics using GRIB
                    input_data_path = self.get_pipeline_path(
                        year, month_str, "downloaded_data"
                    )
                else:  # Metrics using processed NC files
                    input_data_path = self.get_pipeline_path(
                        year, month_str, "processed_data"
                    )

                # Generate paths for index outputs
                index_data_paths = self.get_pipeline_path(year, month_str, "indices"

                # Process the index and handle exceptions
                try:
                    outputs = _calculate_index(
                        index_data_paths,
                        overwrite,
                        input_data_path,
                        self.index_metric,
                        tr_threshold=tr_threshold,
                        hw_min_duration=hw_min_duration,
                        hw_max_gap=hw_max_gap,
                        hw_threshold=hw_threshold,
                    )
                    index_outputs[
                        f"{year}_init{month_str}_valid{self.valid_period_str}"
                    ] = outputs

                except FileNotFoundError:
                    msg = (
                        f"[Index Calculation] Skipped {self.index_metric} for "
                        f"year={year}, month={month_str} — input file not found. "
                        f"Expected: {input_data_path}"
                    )
                    LOGGER.warning(msg)

                except Exception as error:
                    raise RuntimeError(
                        f"Error processing index {self.index_metric} for "
                        f"{year}-{month_str}: {error}"
                    ) from error

        return index_outputs

    ##########  Calculate hazard  ##########

    def save_index_to_hazard(self, overwrite=False):
        """
        Convert the calculated climate index to a CLIMADA Hazard object and save it

        This function reads the monthly aggregated index NetCDF files and converts t
        into a CLIMADA Hazard object. The resulting hazard files are saved in HDF5 f

        Parameters
        ----------
        overwrite : bool, optional
            If True, existing hazard files will be overwritten. Default is False.

        Returns
        -------
        dict
            Dictionary with keys of the form "<year>_init<month>_valid<valid_period>
            and values corresponding to the saved Hazard HDF5 file paths.

        Raises
        ------
        Exception
            If the hazard conversion fails due to missing input files or processing

        Notes
        -----
        The hazard conversion is performed using the `_convert_to_hazard` function.
        The function expects that the index files (monthly NetCDF) have already been
        calculated and saved using `calculate_index()`.

        The resulting Hazard objects follow CLIMADA's internal structure and can be
        used for further risk assessment workflows.
        """
        hazard_outputs = {}

        for year in self.year_list:
            for month_str in self.initiation_month_str:
                LOGGER.info(
                    "Creating hazard for index %s for year %s, initiation month %s."
                    self.index_metric,
                    year,
                    month_str,
                )
                # Get input index file paths and hazard output file paths
                index_data_path = self.get_pipeline_path(year, month_str, "indices")
                    "monthly"
                ]
                hazard_data_path = self.get_pipeline_path(year, month_str, "hazard")
```

```python

                try:
                    # Convert index file to Hazard
                    hazard_outputs[
                        f"{year}_init{month_str}_valid{self.valid_period_str}"
                    ] = _convert_to_hazard(
                        hazard_data_path,
                        overwrite,
                        index_data_path,
                        self.index_metric,
                    )

                except FileNotFoundError:
                    msg = (
                        f"[Hazard Conversion] Skipped {self.index_metric} for year={
                        f"month={month_str} — monthly index file not found."
                    )
                    LOGGER.warning(msg)

                except Exception as error:
                    raise RuntimeError(
                        f"Hazard creation failed for {year}-{month_str}: {error}"
                    ) from error

        return hazard_outputs


##########  Utility Functions  ##########

def handle_overwriting(function):
    """
    Decorator to handle file overwriting during data processing.

    This decorator checks if the target output file(s) already exist and
    whether overwriting is allowed. If the file(s) exist and overwriting
    is disabled, the existing file paths are returned without executing
    the decorated function.

    Parameters
    ----------
    function : callable
        Function to be decorated. Must have the first two arguments:
        - output_file_name : Path or dict of Path
        - overwrite : bool

    Returns
    -------
    callable
        Wrapped function with added file existence check logic.

    Notes
    -----
    - If `output_file_name` is a `Path`, its existence is checked
    - If `output_file_name` is a `Path`, its existence is checked.
    - If `output_file_name` is a `dict` of `Path`, the existence of any file is chec
    - If `overwrite` is False and the file(s) exist, the function is skipped and the
      existing path(s) are returned.
    - The function must accept `overwrite` as the second argument.
    """

    def wrapper(output_file_name, overwrite, *args, **kwargs):
        # if data exists and we do not want to overwrite
        if isinstance(output_file_name, PosixPath):
            if not overwrite and output_file_name.exists():
                LOGGER.info("%s already exists.", output_file_name)
                return output_file_name
        elif isinstance(output_file_name, dict):
            if not overwrite and any(
                path.exists() for path in output_file_name.values()
            ):
                existing_files = [str(path) for path in output_file_name.values()]
                LOGGER.info("One or more files already exist: %s", existing_files)
                return output_file_name

        return function(output_file_name, overwrite, *args, **kwargs)

    return wrapper


##########  Decorated Functions  ##########

@handle_overwriting
def _download_data(
    output_file_name,
    overwrite,
    variables,
    year,
    initiation_month,
    data_format,
    originating_centre,
    system,
    bounds_cds_order,
    leadtimes,
):
    """
    Download seasonal forecast data for a specific year and initiation month.

    This function downloads raw seasonal forecast data from the Copernicus
    Climate Data Store (CDS) based on the specified forecast configuration
    and geographical domain. The data is saved in the specified format and
    location.

    Parameters
    ----------
    output_file_name : Path
        Path to save the downloaded data file.
    overwrite : bool
        If True, existing files will be overwritten. If False and the file exists,
        the download is skipped.
    variables : list of str
        List of variable names to download (e.g., ['tasmax', 'tasmin']).
    year : int
        Year of the forecast initiation.
    initiation_month : int
        Month of the forecast initiation (1-12).
    data_format : str
        File format for the downloaded data ('grib' or 'netcdf').
    originating_centre : str
        Forecast data provider (e.g., 'dwd' for German Weather Service).
    system : str
        Model system identifier (e.g., '21').
    bounds_cds_order : list of float
        Geographical bounding box in CDS order: [north, west, south, east].
    leadtimes : list of int
        List of forecast lead times in hours.

    Returns
    -------
    Path
        Path to the downloaded data file.

    Notes
    -----
    The function uses the `download_data` method from the Copernicus interface modul
    The downloaded data is stored following the directory structure defined by the p
    """
    # Prepare download parameters
    download_params = {
        "data_format": data_format,
        "originating_centre": originating_centre,
        "area": bounds_cds_order,
        "system": system,
        "variable": variables,
        "month": initiation_month,
        "year": year,
        "day": "01",
        "leadtime_hour": leadtimes,
    }

    # Perform download
    downloaded_file = download_data(
        "seasonal-original-single-levels",
        download_params,
        output_file_name,
        overwrite=overwrite,
    )

    return downloaded_file
```

```python

@handle_overwriting
def _process_data(output_file_name, overwrite, input_file_name, variables, data_forma
    """
    Process a downloaded forecast data file into daily NetCDF format.

    This function reads the downloaded forecast data (in GRIB or NetCDF format),
    applies a temporal coarsening operation (aggregation over 4 time steps),
    and saves the resulting daily data as a NetCDF file. For each variable,
    daily mean, maximum, and minimum values are computed.

    Parameters
    ----------
    output_file_name : Path
        Path to save the processed NetCDF file.
    overwrite : bool
        If True, existing processed files will be overwritten. If False and the file
        the processing is skipped.
    input_file_name : Path
        Path to the input downloaded data file.
    variables : list of str
        List of short variable names to process (e.g., ['tasmax', 'tasmin']).
    data_format : str
        Format of the input file ('grib' or 'netcdf').

    Returns
    -------
    Path
        Path to the saved processed NetCDF file.

    Raises
    ------
    FileNotFoundError
        If the input file does not exist.
    Exception
        If an error occurs during data processing.

    Notes
    -----
    The function performs a temporal aggregation by coarsening the data over 4 time
    resulting in daily mean, maximum, and minimum values for each variable.
    The processed data is saved in NetCDF format and can be used for index calculati
    """
    try:
        with xr.open_dataset(
            input_file_name,
            engine="cfgrib" if data_format == "grib" else None,
        ) as input_dataset:
            # Coarsen the data
            ds_mean = input_dataset.coarsen(step=4, boundary="trim").mean()
            ds_max = input_dataset.coarsen(step=4, boundary="trim").max()
            ds_min = input_dataset.coarsen(step=4, boundary="trim").min()

        # Create a new dataset combining mean, max, and min values
        combined_ds = xr.Dataset()
        for var in variables:
            combined_ds[f"{var}_mean"] = ds_mean[var]
            combined_ds[f"{var}_max"] = ds_max[var]
            combined_ds[f"{var}_min"] = ds_min[var]

        # Save the combined dataset to NetCDF
        combined_ds.to_netcdf(str(output_file_name))
        LOGGER.info("Daily file saved to %s", output_file_name)

        return output_file_name

    except FileNotFoundError as error:
        raise FileNotFoundError(
            f"Input file {input_file_name} does not exist. Processing failed."
        ) from error
    except Exception as error:
        raise RuntimeError(
            f"Error during processing for {input_file_name}: {error}"
        ) from error


@handle_overwriting
def _calculate_index(
    output_file_names,
    overwrite,
    input_file_name,
    index_metric,
    tr_threshold=20,
    hw_threshold=27,
    hw_min_duration=3,
    hw_max_gap=0,
):
    """
    Calculate and save climate indices based on the input data.

    Parameters
    ----------
    output_file_names : dict
        Dictionary containing paths for daily, monthly, and stats output files.
    overwrite : bool
        Whether to overwrite existing files.
    input_file_name : Path
        Path to the input file.
    index_metric : str
        Climate index to calculate (e.g., 'HW', 'TR').
    threshold : float, optional
        Threshold for the index calculation (specific to the index type).
    min_duration : int, optional
        Minimum duration for events (specific to the index type).
    max_gap : int, optional
    max_gap : int, optional
        Maximum gap allowed between events (specific to the index type).
    tr_threshold : float, optional
        Threshold for tropical nights (specific to the 'TR' index).

    Returns
    -------
    dict
        Paths to the saved index files.
    """
    # Define output paths
    daily_output_path = output_file_names["daily"]
    monthly_output_path = output_file_names["monthly"]
    stats_output_path = output_file_names["stats"]

    ds_daily, ds_monthly, ds_stats = seasonal_statistics.calculate_heat_indices_metr
        input_file_name,
        index_metric,
        tr_threshold=tr_threshold,
        hw_threshold=hw_threshold,
        hw_min_duration=hw_min_duration,
        hw_max_gap=hw_max_gap,
    )

    # Save outputs
    if ds_daily is not None:
        ds_daily.to_netcdf(daily_output_path)
        LOGGER.info("Saved daily index to %s", daily_output_path)
    if ds_monthly is not None:
        ds_monthly.to_netcdf(monthly_output_path)
        LOGGER.info("Saved monthly index to %s", monthly_output_path)
    if ds_stats is not None:
        ds_stats.to_netcdf(stats_output_path)
        LOGGER.info("Saved stats index to %s", stats_output_path)

    return {
        "daily": daily_output_path,
        "monthly": monthly_output_path,
        "stats": stats_output_path,
    }


@handle_overwriting
def _convert_to_hazard(output_file_name, overwrite, input_file_name, index_metric):
    """
    Convert a climate index file to a CLIMADA Hazard object and save it as HDF5.

    This function reads a processed climate index NetCDF file, converts it to a
    CLIMADA Hazard object, and saves it in HDF5 format. The function supports
    ensemble members and concatenates them into a single Hazard object.

    Parameters
    ----------
```

```
        output_file_name : Path
            Path to save the generated Hazard HDF5 file.
        overwrite : bool
            If True, existing hazard files will be overwritten. If False and the file ex
            the conversion is skipped.
        input_file_name : Path
            Path to the input NetCDF file containing the calculated climate index.
        index_metric : str
            Climate index metric used for hazard creation (e.g., 'HW', 'TR', 'Tmax').

        Returns
        -------
        Path
            Path to the saved Hazard HDF5 file.

        Raises
        ------
        KeyError
            If required variables (e.g., 'step' or index variable) are missing in the da
        Exception
            If the hazard conversion process fails.

        Notes
        -----
        - The function uses `Hazard.from_xarray_raster()` to create Hazard objects
          from the input dataset.
        - If multiple ensemble members are present, individual Hazard objects are
          created for each member and concatenated.
        - The function determines the intensity unit based on the selected index:
            - '%' for relative humidity (RH)
            - 'days' for duration indices (e.g., 'HW', 'TR', 'TX30')
            - '°C' for temperature indices
        """
        try:
            with xr.open_dataset(str(input_file_name)) as input_dataset:
                if "step" not in input_dataset.variables:
                    raise KeyError(
                        f"Missing 'step' variable in dataset for {input_file_name}."
                    )

                input_dataset["step"] = xr.DataArray(
                    [f"{date}-01" for date in input_dataset["step"].values],
                    dims=["step"],
                )
                input_dataset["step"] = pd.to_datetime(input_dataset["step"].values)

                ensemble_members = input_dataset.get("number", [0]).values
                hazards = []

                # Determine intensity unit and variable
                intensity_unit = (
                    "%"
                    if index_metric == "RH"
                    else "days" if index_metric in ["TR", "TX30", "HW"] else "°C"
                )
                intensity_variable = index_metric

                if intensity_variable not in input_dataset.variables:
                    raise KeyError(
                        f"No variable named '{intensity_variable}' in the dataset. "
                        f"Available variables: {list(input_dataset.variables)}"
                    )

                # Create Hazard objects
                for member in ensemble_members:
                    ds_subset = input_dataset.sel(number=member) if "number" in input_da
                    hazard = Hazard.from_xarray_raster(
                        data=ds_subset,
                        hazard_type=index_metric,
                        intensity_unit=intensity_unit,
                        intensity=intensity_variable,
                        coordinate_vars={
                            "event": "step",
                            "longitude": "longitude",
                            "latitude": "latitude",
                        },
                    )
                    hazard.event_name = [
                        f"member{member}" for _ in range(len(hazard.event_name))
                    ]
                    hazards.append(hazard)

                hazard = Hazard.concat(hazards)
                hazard.check()
                hazard.write_hdf5(str(output_file_name))

            LOGGER.info("Hazard file saved to %s.", output_file_name)
            return output_file_name

        except Exception as error:
            raise RuntimeError(
                f"Hazard conversion failed for {input_file_name}: {error}"
            ) from error
```