D. / cli... Type / to search

Code Issues Pull requests Actions Projects Wiki Security

main

climada_seasonal_forecast_sr / create_seasonal_forecast_hazard.py

DahyannAraya Update files a7b428a · 10 minutes ago History

Code Blame 1352 lines (1172 loc) · 47.6 KB Raw

```python
import sys
print(sys.executable)
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


# 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(
```

```
90          self.valid_period_str = "_".join(
91              [f"{month:02d}" for month in self.valid_period]
92          )
93
94          self.index_metric = index_metric
95          self.year_list = year_list
96          self.bounds = bounds
97          self.bounds_str = (
98              f"boundsN{int(self.bounds[0])}_S{int(self.bounds[1])}_"
99              f"E{int(self.bounds[2])}_W{int(self.bounds[3])}"
100             )
101         self.data_format = data_format
102         self.originating_centre = originating_centre
103         self.system = system
104
105         # initialze base directory
106         self.data_out = Path(data_out) if data_out else DATA_OUT
107
108         # Get index specifications
109         index_spec = IndexSpecEnum.get_info(self.index_metric)
110         self.variables = index_spec.variables
111         self.variables_short = [
112             get_short_name_from_variable(var) for var in self.variables
113         ]
114
115     def check_existing_files(
116         self,
117         *,
118         index_metric: str,
119         year: int,
120         initiation_month: str,
121         valid_period: List[str],
122         download_format="grib",
123         print_flag=False,
124     ):
125         """
126         Check whether the forecast data files for the specified parameters exist.
127
128         This function checks the existence of the downloaded raw data, processed data
129         calculated index files, and hazard files for the given forecast configuration
130
131         Parameters
132         ----------
133         index_metric : str
134             Climate index to calculate (e.g., 'HW', 'TR', 'Tmax').
135         year : int
136             Year of the forecast initiation.
137         initiation_month : str
138             Initiation month of the forecast (e.g., 'March', 'April').
139         valid_period : list of str
140             List with start and end month of the valid forecast period, e.g., ['June
141         download_format : str, optional
142             Format of the downloaded data ('grib' or 'netcdf'). Default is 'grib'.
```

```
        print_flag : bool, optional
            If True, prints information about file availability. Default is False.

        Returns
        -------
        str
            Description of which files exist and their locations.

        Raises
        ------
        ValueError
            If valid_period does not contain exactly two months.

        Notes
        -----
        The function checks for the following file types:
        - Downloaded raw data
        - Processed NetCDF data
        - Calculated index NetCDF files (daily, monthly, stats)
        - Hazard HDF5 file

        The file locations are constructed based on the provided parameters and the
        file structure defined in `get_file_path`.
        """
        initiation_month_str = f"{month_name_to_number(initiation_month):02d}"
        valid_period_str = "_".join(
            [f"{month_name_to_number(month):02d}" for month in valid_period]
        )

        (
            downloaded_data_path,
            processed_data_path,
            index_data_paths,
            hazard_data_path,
        ) = [
            self.get_file_path(
                self.data_out,
                self.originating_centre,
                year,
                initiation_month_str,
                valid_period_str,
                data_type,
                index_metric,
                self.bounds_str,
                self.system,
                data_format=download_format,
            )
            for data_type in ["downloaded_data", "processed_data", "indices", "hazar
        ]

        if not downloaded_data_path.exists():
            response = "No downloaded data found for given time periods.\n"
        else:
```

```python
            else:
                response = f"Downloaded data exist at: {downloaded_data_path}\n"
            if not processed_data_path.exists():
                response += "No processed data found for given time periods.\n"
            else:
                response += f"Processed data exist at: {processed_data_path}\n"
            if not any([path.exists() for path in index_data_paths.values()]):
                response += "No index data found for given time periods\n."
            else:
                response += f"Index data exist at: {index_data_paths}\n"
            if not hazard_data_path.exists():
                response += "No hazard data found for given time periods."
            else:
                response += f"Hazard data exist at: {hazard_data_path}"
            if print_flag:
                print(response)
            return response

        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}\n"
            )
            response += (
                "Required variables: "
                f"{', '.join(IndexSpecEnum.get_info(index_metric).variables)}"
            )
            if print_flag:
```

```
            print(response)
        return response

    @staticmethod
    def get_file_path(
        base_dir,
        originating_centre,
        year,
        initiation_month_str,
        valid_period_str,
        data_type,
        index_metric,
        bounds_str,
        system,
        data_format="grib",
    ):
        """Provide file paths for forecast pipeline. For the path tree structure,
        see Notes.

        Parameters
        ----------
        base_dir : _type_
            Base directory where copernicus data and files should be stored. In the p
            not specified differently, CONFIG.hazard.copernicus.seasonal_forecasts.di
        originating_centre : _type_
            Data source (e.g., "dwd").
        year : int or str
            Initiation year.
        initiation_month_str : str
            Initiation month (e.g., '02' or '11').
        valid_period_str : str
            Valid period (e.g., '04_06' or '07_07').
        data_type : str
            Type of the data content. Must be in
            ['downloaded_data', 'processed_data', 'indices', 'hazard'].
        index_metric : str
            Climate index to calculate (e.g., 'HW', 'TR', 'Tmax').
        bounds_str : str
            Spatial bounds as a str, e.g., 'W4_S44_E11_N48'.
        system : _type_
            Model configuration (e.g., "21").
        data_format : str, optional
            Data format ('grib' or 'netcdf').

        Returns
        -------
        pathlib.Path
            Path based on provided parameters.

        Notes
        -----
        The file path will have following structure
        {base_dir}/{originating_centre}/sys{system}/{year}/
```

```
            init{initiation_month_str}/valid{valid_period_str}/
            Depending on the data_type, further subdirectories are created. The paramete
            index_metric and bounds_str are included in the file name.

            Raises
            ------
            ValueError
                If unknown data_type is provided.
            """
            if data_type == "downloaded_data":
                data_type += f"/{data_format}"
            elif data_type == "hazard":
                data_type += f"/{index_metric}"
                data_format = "hdf5"
            elif data_type == "indices":
                data_type += f"/{index_metric}"
                data_format = "nc"
            elif data_type == "processed_data":
                data_format = "nc"
            else:
                raise ValueError(
                    f"Unknown data type {data_type}. Must be in "
                    "['downloaded_data', 'processed_data', 'indices', 'hazard']"
                )

            # prepare parent directory
            sub_dir = (
                f"{base_dir}/{originating_centre}/sys{system}/{year}"
                f"/init{initiation_month_str}/valid{valid_period_str}/{data_type}"
            )

            if data_type.startswith("indices"):
                return {
                    timeframe: Path(
                        f"{sub_dir}/{index_metric}_{bounds_str}_{timeframe}.{data_format
                    )
                    for timeframe in ["daily", "monthly", "stats"]
                }
            return Path(f"{sub_dir}/{index_metric}_{bounds_str}.{data_format}")

        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',
```

```
353
354             Returns
355             -------
356             Path or dict of Path
357                 Path to the requested file(s). For 'indices', returns a dictionary with 
358                 'daily', 'monthly', 'stats'.
359
360             Raises
361             ------
362             ValueError
363                 If unknown data_type is provided.
364
365             Notes
366             -----
367             File structure:
368             {base_dir}/{originating_centre}/sys{system}/{year}/init{init_month}/valid{va
369             /{data_type}
370             """
371
372             file_path = self.get_file_path(
373                 self.data_out,
374                 self.originating_centre,
375                 year,
376                 initiation_month_str,
377                 self.valid_period_str,
378                 data_type,
379                 self.index_metric,
380                 self.bounds_str,
381                 self.system,
382                 self.data_format,
383             )
384
385             # create directory if not existing
386             if data_type == "indices":
387                 file_path["monthly"].parent.mkdir(parents=True, exist_ok=True)
388             else:
389                 file_path.parent.mkdir(parents=True, exist_ok=True)
390
391             return file_path
392
393         ##########  Download and process ##########
394
395         def _download(self, overwrite=False):
396             """
397             Download seasonal forecast data for the specified years and initiation month
398
399             This function downloads the raw forecast data files for each year and initia
400             defined in the instance configuration. The data is downloaded in the specifi
401             ('grib' or 'netcdf') and stored in the configured directory structure.
402
403             Parameters
404             ----------
405             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 di
            structure defined in `get_pipeline_path`. The bounding box is automatically c
            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_peri

                    # 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` functi
        """
        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,
                    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 Exception(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 = {}

        # 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:
                    LOGGER.warning(
                        "File not found for %s-%s. Skipping...", year, month_str
                    )
                except Exception as error:
                    raise Exception(
                        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 a

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

        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")

                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:
                    LOGGER.warning(
                        "Monthly index file not found for %s-%s. Skipping...",
                        year,
                        month_str,
                    )
                except Exception as error:
                    raise Exception(
                        f"Failed to create hazard for {year}-{month_str}: {error}"
                    ) from error

        return hazard_outputs
```

```python

    def plot_forecast_skills(self):
        """
        Access and plot forecast skill data for the handler's parameters,
        filtered by the selected area.
        Raises
        ------
        ValueError
            If the originating_centre is not "dwd".
        ValueError
            If the index_metric is not "Tmax".

        Returns
        -------
        None
            Generates plots for forecast skill metrics based on the handler's parame
            and the selected area.

        """
        # Check if the originating_centre is "dwd"
        if self.originating_centre.lower() != "dwd":
            raise ValueError(
                "Forecast skill metrics are only available for the 'dwd' provider. "
                f"Current provider: {self.originating_centre}"
            )

        # Check if the index_metric is "Tmax"
        if self.index_metric.lower() != "tmax":
            raise ValueError(
                "Forecast skills are only available for the 'Tmax' index. "
                f"Current index: {self.index_metric}"
            )

        # Define the file path pattern for forecast skill data (change for Zenodo wh
        base_path = Path("/Users/daraya/Downloads")
        file_name_pattern = (
            "tasmaxMSESS_subyr_gcfs21_shc{month}-climatology_r1i1p1_1990-2019.nc"
        )

        # Iterate over initiation months and access the corresponding file
        for month_str in self.initiation_month_str:

            # Construct the file name and path
            file_path = base_path / file_name_pattern.format(month=month_str)

            if not file_path.exists():
                LOGGER.warning(
                    "Skill data file for month %s not found: %s",
                    month_str,
                    file_path,
                )
                continue

```

```
                # Load the data using xarray
                try:
                    with xr.open_dataset(file_path) as ds:
                        # Subset the dataset by area bounds
                        west, south, east, north = self.bounds
                        subset_ds = ds.sel(lon=slice(west, east), lat=slice(north, south

                        # Plot each variable
                        variables = [
                            "tasmax_fc_mse",
                            "tasmax_ref_mse",
                            "tasmax_msess",
                            "tasmax_msessSig",
                        ]
                        for var in variables:
                            if var in subset_ds:
                                plt.figure(figsize=(10, 8))
                                ax = plt.axes(projection=ccrs.PlateCarree())

                                # Adjust color scale to improve clarity
                                vmin = subset_ds[var].quantile(0.05).item()
                                vmax = subset_ds[var].quantile(0.95).item()

                                plot_handle = (
                                    subset_ds[var]
                                    .isel(time=0)
                                    .plot(
                                        ax=ax,
                                        cmap="coolwarm",
                                        vmin=vmin,
                                        vmax=vmax,
                                        add_colorbar=False,
                                    )
                                )

                                cbar = plt.colorbar(
                                    plot_handle, ax=ax, orientation="vertical", pad=0.1,
                                )
                                cbar.set_label(var, fontsize=10)

                                ax.set_extent(
                                    [west, east, south, north], crs=ccrs.PlateCarree()
                                )
                                ax.add_feature(cfeature.BORDERS, linestyle=":")
                                ax.add_feature(cfeature.COASTLINE)
                                ax.add_feature(cfeature.LAND, edgecolor="black", alpha=0
                                ax.gridlines(draw_labels=True, crs=ccrs.PlateCarree())

                                plt.title(
                                    f"{var} for month {month_str},  {self.bounds_str}"
                                )
                                plt.show()
```

```python
                            else:
                                LOGGER.warning(
                                    "Variable %s not found in dataset for month %s.",
                                    var,
                                    month_str,
                                )
                    except Exception as error:
                        raise RuntimeError(
                            f"Failed to load or process data for month {month_str}: {error}"
                        ) from error


# ----- Utility Functions -----
# Utility function for month name to number conversion (if not already defined)


def month_name_to_number(month):
    """
    Convert a month name or number to its corresponding integer value.

    Accepts either an integer (1-12), full month name (e.g., 'March'),
    or abbreviated month name (e.g., 'Mar') and returns the corresponding
    month number (1-12).

    Parameters
    ----------
    month : int or str
        Month as an integer (1-12) or as a string (full or abbreviated month name).

    Returns
    -------
    int
        Month as an integer in the range 1-12.

    Raises
    ------
    ValueError
        If the input month is invalid, empty, or outside the valid range.
    """
    if isinstance(month, int):  # Already a number
        if 1 <= month <= 12:
            return month
        else:
            raise ValueError("Month number must be between 1 and 12.")
    if isinstance(month, str):
        if not month.strip():
            raise ValueError("Month cannot be empty.")  # e.g. "" or "   "
        month = month.capitalize()  # Ensure consistent capitalization
        if month in calendar.month_name:
            return list(calendar.month_name).index(month)
        elif month in calendar.month_abbr:
            return list(calendar.month_abbr).index(month)
    raise ValueError(f"Invalid month input: {month}")
```

```python


def calculate_leadtimes(year, initiation_month, valid_period):
    """
    Calculate lead times in hours for a forecast period based on initiation and vali

    This function computes a list of lead times (in hours) for a seasonal forecast, 
    from the initiation month to the end of the valid period. The lead times are gen
    in 6-hour steps, following the standard forecast output intervals.

    Parameters
    ----------
    year : int
        Year of the forecast initiation.
    initiation_month : int or str
        Initiation month of the forecast, as integer (1-12) or month name (e.g., 'Ma
    valid_period : list of int or str
        List containing the start and end month of the valid period, either as integ
        or month names (e.g., ['June', 'August']). Must contain exactly two elements

    Returns
    -------
    list of int
        List of lead times in hours, sorted and spaced by 6 hours.

    Raises
    ------
    ValueError
        If initiation month or valid period months are invalid or reversed.
    Exception
        For general errors during lead time calculation.

    Notes
    -----
    - The valid period may extend into the following year if the valid months are af
    - Lead times are calculated relative to the initiation date.
    - Each lead time corresponds to a 6-hour forecast step.

    Example:
    ---------
    If the forecast is initiated in **December 2022** and the valid period is **Janu
    to February 2023**,
    the function will:
    - Recognize that the forecast extends into the next year (2023).
    - Compute lead times starting from **December 1, 2022** (0 hours) to **February 
    - Generate lead times in 6-hour intervals, covering the entire forecast period f
    December 2022 through February 2023.
    """

    # Convert initiation month to numeric if it is a string
    if isinstance(initiation_month, str):
        initiation_month = month_name_to_number(initiation_month)
```

```python

        # Convert valid_period to numeric
        valid_period = [
            month_name_to_number(month) if isinstance(month, str) else month
            for month in valid_period
        ]

        # We expect valid_period = [start, end]
        start_month, end_month = valid_period

        # Immediately check for reversed period
        if end_month < start_month:
            raise ValueError(
                "Reversed valid_period detected. The forecast cannot be called with "
                f"an end month ({end_month}) that is before the start month ({start_montl
            )

        # compute years of valid period
        valid_years = np.array([year, year])
        if initiation_month > valid_period[0]:  # forecast for next year
            valid_years += np.array([1, 1])
        if valid_period[1] < valid_period[0]:  # forecast including two different years
            valid_years[1] += 1

        # Reference starting date for initiation
        initiation_date = date(year, initiation_month, 1)
        valid_period_start = date(valid_years[0], valid_period[0], 1)
        valid_period_end = date(
            valid_years[1],
            valid_period[1],
            calendar.monthrange(valid_years[1], valid_period[1])[1],
        )

        return list(
            range(
                (valid_period_start - initiation_date).days * 24,
                (valid_period_end - initiation_date).days * 24 + 24,
                6,
            )
        )


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 `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


    @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 s
        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 ds:
                # Coarsen the data
                ds_mean = ds.coarsen(step=4, boundary="trim").mean()
                ds_max = ds.coarsen(step=4, boundary="trim").max()
                ds_min = ds.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 Exception(
                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
        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_metri
        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 ds:
            if "step" not in ds.variables:
                raise KeyError(
                    f"Missing 'step' variable in dataset for {input_file_name}."
                )
```

```python

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

                ensemble_members = ds.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 ds.variables:
                    raise KeyError(
                        f"No variable named '{intensity_variable}' in the dataset. "
                        f"Available variables: {list(ds.variables)}"
                    )

                # Create Hazard objects
                for member in ensemble_members:
                    ds_subset = ds.sel(number=member) if "number" in ds.dims else ds
                    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 Exception(
                f"Failed to convert {input_file_name} to hazard: {error}"
```

```
            ) from error
```