
 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



```
1  import sys
2  print(sys.executable)
3  import calendar
4  import logging
5  from datetime import date
6  from pathlib import Path, PosixPath
7  from typing import List
8
9  import cartopy.crs as ccrs
10 import cartopy.feature as cfeature
11 import matplotlib.pyplot as plt
12 import numpy as np
13 import pandas as pd
14 import xarray as xr
15
16 from climada import CONFIG
17 from climada.hazard import Hazard
18 from climada_petals.hazard.copernicus_interface.downloader import download_data
19 from climada_petals.hazard.copernicus_interface.index_definitions import (
20     IndexSpecEnum,
21     get_short_name_from_variable,
22 )
23 import climada_petals.hazard.copernicus_interface.seasonal_statistics as seasonal_st
24
25
26 # set path to store data
27 DATA_OUT = CONFIG.hazard.copernicus.seasonal_forecasts.dir()
28 LOGGER = logging.getLogger(__name__)
29
30
31 # ----- Main Class -----
32 class SeasonalForecast:
33     """
34     Class for managing the download, processing, and analysis of seasonal climate fo
35     """
36
37     def __init__(
```

```

38         self,
39         index_metric,
40         year_list,
41         forecast_period,
42         initiation_month,
43         bounds,
44         data_format,
45         originating_centre,
46         system,
47         data_out=None,
48     ):
49         """
50         Initialize the SeasonalForecast instance with user-defined parameters for in
51
52         Parameters
53         -----
54         index_metric : str
55             Climate index to calculate (e.g., "HW", "TR", "Tmax").
56         year_list : list of int
57             List of years for which data should be downloaded and processed.
58         lead_time_months : list of str or int
59             List specifying the start and end month (given as integers or strings)
60             of the valid forecast period. Must contain exactly two elements.
61         initiation_month : list of str
62             List of initiation months for the forecast (e.g., ["March", "April"]).
63         bounds : list of float
64             Bounding box values in EPSG 4326 format: (min_lon, min_lat, max_lon, max
65         data_format : str
66             Format of the downloaded data. Either "grib" or "netcdf".
67         originating_centre : str
68             Data provider (e.g., "dwd").
69         system : str
70             Forecast system configuration (e.g., "21").
71         data_out : pathlib.Path, optional
72             Output directory for storing downloaded and processed data. If None,
73             uses a default directory specified in the configuration.
74
75         Raises
76         -----
77         ValueError
78             If the valid period does not contain exactly two months.
79         """
80         # initiate initiation month, valid period, and leadtimes
81         valid_period = forecast_period
82         if not isinstance(initiation_month, list):
83             initiation_month = [initiation_month]
84         if not isinstance(valid_period, list) or len(valid_period) != 2:
85             raise ValueError("Valid period must be a list of two months.")
86         self.initiation_month_str = [
87             f"{month_name_to_number(month):02d}" for month in initiation_month
88         ]
89         self.valid_period = [month_name_to_number(month) for month in valid_period]
90         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     # initialize 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'.

```

```

143     print_flag : bool, optional
144         If True, prints information about file availability. Default is False.
145
146     Returns
147     -----
148     str
149         Description of which files exist and their locations.
150
151     Raises
152     -----
153     ValueError
154         If valid_period does not contain exactly two months.
155
156     Notes
157     -----
158     The function checks for the following file types:
159     - Downloaded raw data
160     - Processed NetCDF data
161     - Calculated index NetCDF files (daily, monthly, stats)
162     - Hazard HDF5 file
163
164     The file locations are constructed based on the provided parameters and the
165     file structure defined in `get_file_path`.
166     """
167     initiation_month_str = f"{month_name_to_number(initiation_month):02d}"
168     valid_period_str = "_".join(
169         [f"{month_name_to_number(month):02d}" for month in valid_period]
170     )
171
172     (
173         downloaded_data_path,
174         processed_data_path,
175         index_data_paths,
176         hazard_data_path,
177     ) = [
178         self.get_file_path(
179             self.data_out,
180             self.Originating_centre,
181             year,
182             initiation_month_str,
183             valid_period_str,
184             data_type,
185             index_metric,
186             self.bounds_str,
187             self.system,
188             data_format=download_format,
189         )
190         for data_type in ["downloaded_data", "processed_data", "indices", "hazard"]
191     ]
192
193     if not downloaded_data_path.exists():
194         response = "No downloaded data found for given time periods.\n"
195     else:

```

```

170         else:
171             response = f"Downloaded data exist at: {downloaded_data_path}\n"
172         if not processed_data_path.exists():
173             response += "No processed data found for given time periods.\n"
174         else:
175             response += f"Processed data exist at: {processed_data_path}\n"
176         if not any([path.exists() for path in index_data_paths.values()]):
177             response += "No index data found for given time periods.\n"
178         else:
179             response += f"Index data exist at: {index_data_paths}\n"
180         if not hazard_data_path.exists():
181             response += "No hazard data found for given time periods."
182         else:
183             response += f"Hazard data exist at: {hazard_data_path}"
184         if print_flag:
185             print(response)
186         return response
187
188
189 ✓ def explain_index(self, index_metric=None, print_flag=False):
190     """
191     Retrieve and display information about a specific climate index.
192
193     This function provides an explanation and the required input variables for
194     the selected climate index. If no index is provided, the instance's
195     `index_metric` is used.
196
197     Parameters
198     -----
199     index_metric : str, optional
200         Climate index to explain (e.g., 'HW', 'TR', 'Tmax'). If None, uses the
201         instance's index_metric.
202     print_flag : bool, optional
203         If True, prints the explanation. Default is False.
204
205     Returns
206     -----
207     str
208         Text description of the index explanation and required input variables.
209
210     Notes
211     -----
212     The index information is retrieved from `IndexSpecEnum.get_info`.
213     """
214     index_metric = index_metric or self.index_metric
215     response = (
216         f"Explanation for {index_metric}: "
217         f"{IndexSpecEnum.get_info(index_metric).explanation}\n"
218     )
219     response += (
220         "Required variables: "
221         f"{', '.join(IndexSpecEnum.get_info(index_metric).variables)}"
222     )
223     if print_flag:

```

```

248         print(response)
249     return response
250
251     @staticmethod
252     def get_file_path(
253         base_dir,
254         originating_centre,
255         year,
256         initiation_month_str,
257         valid_period_str,
258         data_type,
259         index_metric,
260         bounds_str,
261         system,
262         data_format="grib",
263     ):
264         """Provide file paths for forecast pipeline. For the path tree structure,
265         see Notes.
266
267         Parameters
268         -----
269         base_dir : _type_
270             Base directory where copernicus data and files should be stored. In the
271             not specified differently, CONFIG.hazard.copernicus.seasonal_forecasts.d
272         originating_centre : _type_
273             Data source (e.g., "dwd").
274         year : int or str
275             Initiation year.
276         initiation_month_str : str
277             Initiation month (e.g., '02' or '11').
278         valid_period_str : str
279             Valid period (e.g., '04_06' or '07_07').
280         data_type : str
281             Type of the data content. Must be in
282             ['downloaded_data', 'processed_data', 'indices', 'hazard'].
283         index_metric : str
284             Climate index to calculate (e.g., 'HW', 'TR', 'Tmax').
285         bounds_str : str
286             Spatial bounds as a str, e.g., 'W4_S44_E11_N48'.
287         system : _type_
288             Model configuration (e.g., "21").
289         data_format : str, optional
290             Data format ('grib' or 'netcdf').
291
292         Returns
293         -----
294         pathlib.Path
295             Path based on provided parameters.
296
297         Notes
298         -----
299         The file path will have following structure
300         {base_dir}/{oriainating centre}/svs{svsystem}/{vear}/

```

```

301         init{initiation_month_str}/valid{valid_period_str}/
302         Depending on the data_type, further subdirectories are created. The parameter
303         index_metric and bounds_str are included in the file name.
304
305     Raises
306     -----
307     ValueError
308         If unknown data_type is provided.
309     """
310     if data_type == "downloaded_data":
311         data_type += f"/{data_format}"
312     elif data_type == "hazard":
313         data_type += f"/{index_metric}"
314         data_format = "hdf5"
315     elif data_type == "indices":
316         data_type += f"/{index_metric}"
317         data_format = "nc"
318     elif data_type == "processed_data":
319         data_format = "nc"
320     else:
321         raise ValueError(
322             f"Unknown data type {data_type}. Must be in "
323             "['downloaded_data', 'processed_data', 'indices', 'hazard']"
324         )
325
326     # prepare parent directory
327     sub_dir = (
328         f"{base_dir}/{originating_centre}/sys{system}/{year}"
329         f"/init{initiation_month_str}/valid{valid_period_str}/{data_type}"
330     )
331
332     if data_type.startswith("indices"):
333         return {
334             timeframe: Path(
335                 f"{sub_dir}/{index_metric}_{bounds_str}_{timeframe}.{data_format}"
336             )
337             for timeframe in ["daily", "monthly", "stats"]
338         }
339     return Path(f"{sub_dir}/{index_metric}_{bounds_str}.{data_format}")
340
341     ✓ def get_pipeline_path(self, year, initiation_month_str, data_type):
342         """
343         Provide (and possibly create) file paths for forecast pipeline.
344
345         Parameters
346         -----
347         year : int
348             Year of the forecast initiation.
349         init_month : str
350             Initiation month as two-digit string (e.g., '03' for March).
351         data_type : str
352             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

```



```

406         If True, existing downloaded files will be overwritten. Default is False
407
408     Returns
409     -----
410     dict
411         Dictionary with keys of the form "<year>_init<month>_valid<valid_period>"
412         and values corresponding to the downloaded data file paths.
413
414     Notes
415     -----
416     The data is downloaded using the `_download_data` function and follows the d
417     structure defined in `get_pipeline_path`. The bounding box is automatically c
418     to CDS (Climate Data Store) format before download.
419     """
420     output_files = {}
421     bounds_cds_order = [
422         self.bounds[3],
423         *self.bounds[:3],
424     ] # convert bounds to CDS order
425     for year in self.year_list:
426         for month_str in self.initiation_month_str:
427             leadtimes = calculate_leadtimes(year, int(month_str), self.valid_peri
428
429             # Generate output file name
430             downloaded_data_path = self.get_pipeline_path(
431                 year, month_str, "downloaded_data"
432             )
433
434             output_files[f"{year}_init{month_str}_valid{self.valid_period_str}"]
435                 _download_data(
436                     downloaded_data_path,
437                     overwrite,
438                     self.variables,
439                     year,
440                     month_str,
441                     self.data_format,
442                     self.originating_centre,
443                     self.system,
444                     bounds_cds_order,
445                     leadtimes,
446                 )
447             )
448
449     return output_files
450
451     ✓ def _process(self, overwrite=False):
452         """
453         Process the downloaded forecast data into daily NetCDF format.
454
455         This function processes the raw downloaded data files into a standardized
456         daily NetCDF format, applying basic aggregation operations (mean, max, min).
457         The processed files are saved in the configured output directory.
458     """

```

```

458
459     Parameters
460     -----
461     overwrite : bool, optional
462         If True, existing processed files will be overwritten. Default is False.
463
464     Returns
465     -----
466     dict
467         Dictionary with keys of the form "<year>_init<month>_valid<valid_period>"
468         and values corresponding to the processed NetCDF file paths.
469
470     Notes
471     -----
472     The processing applies a daily coarsening operation and aggregates the data.
473     The processed data is saved in NetCDF format in the directory defined by
474     `get_pipeline_path`. Processing is performed using the `_process_data` function.
475     """
476     processed_files = {}
477     for year in self.year_list:
478         for month_str in self.initiation_month_str:
479             # Locate input file name
480             downloaded_data_path = self.get_pipeline_path(
481                 year, month_str, "downloaded_data"
482             )
483             # Generate output file name
484             processed_data_path = self.get_pipeline_path(
485                 year, month_str, "processed_data"
486             )
487
488             processed_files[
489                 f"{year}_init{month_str}_valid{self.valid_period_str}"
490             ] = _process_data(
491                 processed_data_path,
492                 overwrite,
493                 downloaded_data_path,
494                 self.variables_short,
495                 self.data_format,
496             )
497
498     return processed_files
499
500 ✓ def download_and_process_data(self, overwrite=False):
501     """
502     Download and process seasonal climate forecast data.
503
504     This function performs the complete data pipeline by first downloading
505     the raw forecast data for the specified years and initiation months,
506     and then processing the downloaded data into a daily NetCDF format.
507
508     Parameters
509     -----
510     overwrite : bool, optional

```

```

511         If True, existing downloaded and processed files will be overwritten. De
512
513     Returns
514     -----
515     dict
516         Dictionary containing two keys:
517         - "downloaded_data": dict with file paths to downloaded raw data.
518         - "processed_data": dict with file paths to processed NetCDF data.
519
520     Raises
521     -----
522     Exception
523         If an error occurs during download or processing, such as invalid input ,
524         or file system issues.
525
526     Notes
527     -----
528     This is a high-level method that internally calls `_download()` and `_proces
529     The file structure and naming follow the configuration defined in `get_pipel
530     """
531
532     # Call high-level methods for downloading and processing
533     created_files = {}
534     try:
535         # 1) Attempt downloading data
536         created_files["downloaded_data"] = self._download(overwrite=overwrite)
537         # 2) Attempt processing data
538         created_files["processed_data"] = self._process(overwrite=overwrite)
539     except Exception as error:
540         # Catch reversed valid_period or any other ValueError from calculate_lea
541         raise Exception(f"Download/process aborted: {error}") from error
542
543     return created_files
544
545     ##### Calculate index #####
546
547     def calculate_index(
548         self,
549         overwrite=False,
550         hw_threshold=27,
551         hw_min_duration=3,
552         hw_max_gap=0,
553         tr_threshold=20,
554     ):
555         """
556         Calculate the specified climate index based on the downloaded forecast data.
557
558         This function processes the downloaded or processed forecast data to compute
559         the selected climate index (e.g., Heatwave days, Tropical Nights) according
560         to the parameters defined for the index.
561
562         Parameters

```

```

503 -----
564 overwrite : bool, optional
565     If True, existing index files will be overwritten. Default is False.
566 hw_threshold : float, optional
567     Temperature threshold for heatwave days index calculation. Default is 27
568 hw_min_duration : int, optional
569     Minimum duration (in days) of consecutive conditions for a heatwave event
570 hw_max_gap : int, optional
571     Maximum allowable gap (in days) between conditions to still
572     consider as a single heatwave event. Default is 0.
573 tr_threshold : float, optional
574     Temperature threshold for tropical nights index calculation. Default is 27
575
576 Returns
577 -----
578 dict
579     Dictionary with keys of the form "<year>_init<month>_valid<valid_period>"
580     and values corresponding to the output NetCDF index files (daily, monthly)
581
582 Raises
583 -----
584 Exception
585     If index calculation fails due to missing files or processing errors.
586
587 Notes
588 -----
589 The input files used depend on the index:
590 - For 'TX30', 'TR', and 'HW', the raw downloaded GRIB data is used.
591 - For other indices, the processed NetCDF data is used.
592
593 The calculation is performed using the `_calculate_index` function and results
594 are saved in the configured output directory structure.
595 """
596 index_outputs = {}
597
598 # Iterate over each year and initiation month
599 for year in self.year_list:
600     for month_str in self.initiation_month_str:
601         LOGGER.info(
602             "Processing index %s for year %s, initiation month %s.",
603             self.index_metric,
604             year,
605             month_str,
606         )
607
608         # Determine the input file based on index type
609         if self.index_metric in ["TX30", "TR", "HW"]: # Metrics using GRIB
610             input_data_path = self.get_pipeline_path(
611                 year, month_str, "downloaded_data"
612             )
613         else: # Metrics using processed NC files
614             input_data_path = self.get_pipeline_path(
615                 year, month_str, "processed_data"

```

```

616         )
617
618         # Generate paths for index outputs
619         index_data_paths = self.get_pipeline_path(year, month_str, "indices")
620
621         # Process the index and handle exceptions
622         try:
623             outputs = _calculate_index(
624                 index_data_paths,
625                 overwrite,
626                 input_data_path,
627                 self.index_metric,
628                 tr_threshold=tr_threshold,
629                 hw_min_duration=hw_min_duration,
630                 hw_max_gap=hw_max_gap,
631                 hw_threshold=hw_threshold,
632             )
633             index_outputs[
634                 f"{year}_init{month_str}_valid{self.valid_period_str}"
635             ] = outputs
636
637         except FileNotFoundError:
638             LOGGER.warning(
639                 "File not found for %s-%s. Skipping...", year, month_str
640             )
641         except Exception as error:
642             raise Exception(
643                 f"Error processing index {self.index_metric} for "
644                 f"{year}-{month_str}: {error}"
645             ) from error
646
647         return index_outputs
648
649     ##### Calculate hazard #####
650
651     ✓ def save_index_to_hazard(self, overwrite=False):
652         """
653         Convert the calculated climate index to a CLIMADA Hazard object and save it
654
655         This function reads the monthly aggregated index NetCDF files and converts them
656         into a CLIMADA Hazard object. The resulting hazard files are saved in HDF5 format.
657
658         Parameters
659         -----
660         overwrite : bool, optional
661             If True, existing hazard files will be overwritten. Default is False.
662
663         Returns
664         -----
665         dict
666             Dictionary with keys of the form "<year>_init<month>_valid<valid_period>"
667             and values corresponding to the saved Hazard HDF5 file paths.
668
669

```

```

669         Raises
670         -----
671         Exception
672             If the hazard conversion fails due to missing input files or processing
673
674         Notes
675         -----
676         The hazard conversion is performed using the `_convert_to_hazard` function.
677         The function expects that the index files (monthly NetCDF) have already been
678         calculated and saved using `calculate_index()`.
679
680         The resulting Hazard objects follow CLIMADA's internal structure and can be
681         used for further risk assessment workflows.
682         """
683         hazard_outputs = {}
684
685         for year in self.year_list:
686             for month_str in self.initiation_month_str:
687                 LOGGER.info(
688                     "Creating hazard for index %s for year %s, initiation month %s."
689                     self.index_metric,
690                     year,
691                     month_str,
692                 )
693                 # Get input index file paths and hazard output file paths
694                 index_data_path = self.get_pipeline_path(year, month_str, "indices")
695                 "monthly"
696             ]
697             hazard_data_path = self.get_pipeline_path(year, month_str, "hazard")
698
699             try:
700                 # Convert index file to Hazard
701                 hazard_outputs[
702                     f"{year}_init{month_str}_valid{self.valid_period_str}"
703                 ] = _convert_to_hazard(
704                     hazard_data_path,
705                     overwrite,
706                     index_data_path,
707                     self.index_metric,
708                 )
709             except FileNotFoundError:
710                 LOGGER.warning(
711                     "Monthly index file not found for %s-%s. Skipping...",
712                     year,
713                     month_str,
714                 )
715             except Exception as error:
716                 raise Exception(
717                     f"Failed to create hazard for {year}-{month_str}: {error}"
718                 ) from error
719
720         return hazard_outputs

```

```

721
722 ✓ def plot_forecast_skills(self):
723     """
724     Access and plot forecast skill data for the handler's parameters,
725     filtered by the selected area.
726     Raises
727     -----
728     ValueError
729         If the originating_centre is not "dwd".
730     ValueError
731         If the index_metric is not "Tmax".
732
733     Returns
734     -----
735     None
736         Generates plots for forecast skill metrics based on the handler's parameters
737         and the selected area.
738
739     """
740     # Check if the originating_centre is "dwd"
741     if self.Originating_Centre.lower() != "dwd":
742         raise ValueError(
743             "Forecast skill metrics are only available for the 'dwd' provider. "
744             f"Current provider: {self.Originating_Centre}"
745         )
746
747     # Check if the index_metric is "Tmax"
748     if self.index_metric.lower() != "tmax":
749         raise ValueError(
750             "Forecast skills are only available for the 'Tmax' index. "
751             f"Current index: {self.index_metric}"
752         )
753
754     # Define the file path pattern for forecast skill data (change for Zenodo when available)
755     base_path = Path("/Users/daraya/Downloads")
756     file_name_pattern = (
757         "tasmaxMSESS_subyr_gcfs21_shc{month}-climatology_r1i1p1_1990-2019.nc"
758     )
759
760     # Iterate over initiation months and access the corresponding file
761     for month_str in self.initiation_month_str:
762
763         # Construct the file name and path
764         file_path = base_path / file_name_pattern.format(month=month_str)
765
766         if not file_path.exists():
767             LOGGER.warning(
768                 "Skill data file for month %s not found: %s",
769                 month_str,
770                 file_path,
771             )
772             continue
773

```

```

774         # Load the data using xarray
775         try:
776             with xr.open_dataset(file_path) as ds:
777                 # Subset the dataset by area bounds
778                 west, south, east, north = self.bounds
779                 subset_ds = ds.sel(lon=slice(west, east), lat=slice(north, south
780
781                 # Plot each variable
782                 variables = [
783                     "tasmax_fc_mse",
784                     "tasmax_ref_mse",
785                     "tasmax_msess",
786                     "tasmax_msessSig",
787                 ]
788                 for var in variables:
789                     if var in subset_ds:
790                         plt.figure(figsize=(10, 8))
791                         ax = plt.axes(projection=ccrs.PlateCarree())
792
793                         # Adjust color scale to improve clarity
794                         vmin = subset_ds[var].quantile(0.05).item()
795                         vmax = subset_ds[var].quantile(0.95).item()
796
797                         plot_handle = (
798                             subset_ds[var]
799                             .isel(time=0)
800                             .plot(
801                                 ax=ax,
802                                 cmap="coolwarm",
803                                 vmin=vmin,
804                                 vmax=vmax,
805                                 add_colorbar=False,
806                             )
807                         )
808
809                         cbar = plt.colorbar(
810                             plot_handle, ax=ax, orientation="vertical", pad=0.1,
811                         )
812                         cbar.set_label(var, fontsize=10)
813
814                         ax.set_extent(
815                             [west, east, south, north], crs=ccrs.PlateCarree()
816                         )
817                         ax.add_feature(cfeature.BORDERS, linestyle=":")
818                         ax.add_feature(cfeature.COASTLINE)
819                         ax.add_feature(cfeature.LAND, edgecolor="black", alpha=0
820                         ax.gridlines(draw_labels=True, crs=ccrs.PlateCarree())
821
822                         plt.title(
823                             f"{var} for month {month_str}, {self.bounds_str}"
824                         )
825                         plt.show()

```



```

826         else:
827             LOGGER.warning(
828                 "Variable %s not found in dataset for month %s.",
829                 var,
830                 month_str,
831             )
832     except Exception as error:
833         raise RuntimeError(
834             f"Failed to load or process data for month {month_str}: {error}"
835         ) from error
836
837
838 # ----- Utility Functions -----
839 # Utility function for month name to number conversion (if not already defined)
840
841
842 ✓ def month_name_to_number(month):
843     """
844     Convert a month name or number to its corresponding integer value.
845
846     Accepts either an integer (1-12), full month name (e.g., 'March'),
847     or abbreviated month name (e.g., 'Mar') and returns the corresponding
848     month number (1-12).
849
850     Parameters
851     -----
852     month : int or str
853         Month as an integer (1-12) or as a string (full or abbreviated month name).
854
855     Returns
856     -----
857     int
858         Month as an integer in the range 1-12.
859
860     Raises
861     -----
862     ValueError
863         If the input month is invalid, empty, or outside the valid range.
864     """
865     if isinstance(month, int): # Already a number
866         if 1 <= month <= 12:
867             return month
868         else:
869             raise ValueError("Month number must be between 1 and 12.")
870     if isinstance(month, str):
871         if not month.strip():
872             raise ValueError("Month cannot be empty.") # e.g. "" or " "
873         month = month.capitalize() # Ensure consistent capitalization
874         if month in calendar.month_name:
875             return list(calendar.month_name).index(month)
876         elif month in calendar.month_abbr:
877             return list(calendar.month_abbr).index(month)
878         raise ValueError(f"Invalid month input: {month}")

```

```

879
880
881 ✓ def calculate_leadtimes(year, initiation_month, valid_period):
882     """
883     Calculate lead times in hours for a forecast period based on initiation and valid
884
885     This function computes a list of lead times (in hours) for a seasonal forecast,
886     from the initiation month to the end of the valid period. The lead times are generated
887     in 6-hour steps, following the standard forecast output intervals.
888
889     Parameters
890     -----
891     year : int
892         Year of the forecast initiation.
893     initiation_month : int or str
894         Initiation month of the forecast, as integer (1-12) or month name (e.g., 'March').
895     valid_period : list of int or str
896         List containing the start and end month of the valid period, either as integers
897         or month names (e.g., ['June', 'August']). Must contain exactly two elements.
898
899     Returns
900     -----
901     list of int
902         List of lead times in hours, sorted and spaced by 6 hours.
903
904     Raises
905     -----
906     ValueError
907         If initiation month or valid period months are invalid or reversed.
908     Exception
909         For general errors during lead time calculation.
910
911     Notes
912     -----
913     - The valid period may extend into the following year if the valid months are after
914     - Lead times are calculated relative to the initiation date.
915     - Each lead time corresponds to a 6-hour forecast step.
916
917     Example:
918     -----
919     If the forecast is initiated in **December 2022** and the valid period is **January
920     to February 2023**,
921     the function will:
922     - Recognize that the forecast extends into the next year (2023).
923     - Compute lead times starting from **December 1, 2022** (0 hours) to **February 1, 2023**.
924     - Generate lead times in 6-hour intervals, covering the entire forecast period from
925     December 2022 through February 2023.
926     """
927
928     # Convert initiation month to numeric if it is a string
929     if isinstance(initiation_month, str):
930         initiation_month = month_name_to_number(initiation_month)
931
932

```

```

931
932     # Convert valid_period to numeric
933     valid_period = [
934         month_name_to_number(month) if isinstance(month, str) else month
935         for month in valid_period
936     ]
937
938     # We expect valid_period = [start, end]
939     start_month, end_month = valid_period
940
941     # Immediately check for reversed period
942     if end_month < start_month:
943         raise ValueError(
944             "Reversed valid_period detected. The forecast cannot be called with "
945             f"an end month ({end_month}) that is before the start month ({start_month})"
946         )
947
948     # compute years of valid period
949     valid_years = np.array([year, year])
950     if initiation_month > valid_period[0]: # forecast for next year
951         valid_years += np.array([1, 1])
952     if valid_period[1] < valid_period[0]: # forecast including two different years
953         valid_years[1] += 1
954
955     # Reference starting date for initiation
956     initiation_date = date(year, initiation_month, 1)
957     valid_period_start = date(valid_years[0], valid_period[0], 1)
958     valid_period_end = date(
959         valid_years[1],
960         valid_period[1],
961         calendar.monthrange(valid_years[1], valid_period[1])[1],
962     )
963
964     return list(
965         range(
966             (valid_period_start - initiation_date).days * 24,
967             (valid_period_end - initiation_date).days * 24 + 24,
968             6,
969         )
970     )
971
972
973 ✓ def handle_overwriting(function):
974     """
975     Decorator to handle file overwriting during data processing.
976
977     This decorator checks if the target output file(s) already exist and
978     whether overwriting is allowed. If the file(s) exist and overwriting
979     is disabled, the existing file paths are returned without executing
980     the decorated function.
981
982     Parameters
983     -----

```

```

984     function : callable
985         Function to be decorated. Must have the first two arguments:
986         - output_file_name : Path or dict of Path
987         - overwrite : bool
988
989     Returns
990     -----
991     callable
992         Wrapped function with added file existence check logic.
993
994     Notes
995     -----
996     - If `output_file_name` is a `Path`, its existence is checked.
997     - If `output_file_name` is a `dict` of `Path`, the existence of any file is checked.
998     - If `overwrite` is False and the file(s) exist, the function is skipped and the
999       existing path(s) are returned.
1000     - The function must accept `overwrite` as the second argument.
1001     """
1002
1003     def wrapper(output_file_name, overwrite, *args, **kwargs):
1004         # if data exists and we do not want to overwrite
1005         if isinstance(output_file_name, PosixPath):
1006             if not overwrite and output_file_name.exists():
1007                 LOGGER.info("%s already exists.", output_file_name)
1008                 return output_file_name
1009             elif isinstance(output_file_name, dict):
1010                 if not overwrite and any(
1011                     path.exists() for path in output_file_name.values()
1012                 ):
1013                     existing_files = [str(path) for path in output_file_name.values()]
1014                     LOGGER.info("One or more files already exist: %s", existing_files)
1015                     return output_file_name
1016
1017             return function(output_file_name, overwrite, *args, **kwargs)
1018
1019     return wrapper
1020
1021 # ----- Decorated Functions -----
1022
1023
1024
1025 @handle_overwriting
1026 def _download_data(
1027     output_file_name,
1028     overwrite,
1029     variables,
1030     year,
1031     initiation_month,
1032     data_format,
1033     originating_centre,
1034     system,
1035     bounds_cds_order,
1036     ...

```

```

1036         leadtimes,
1037     ):
1038         """
1039         Download seasonal forecast data for a specific year and initiation month.
1040
1041         This function downloads raw seasonal forecast data from the Copernicus
1042         Climate Data Store (CDS) based on the specified forecast configuration
1043         and geographical domain. The data is saved in the specified format and
1044         location.
1045
1046         Parameters
1047         -----
1048         output_file_name : Path
1049             Path to save the downloaded data file.
1050         overwrite : bool
1051             If True, existing files will be overwritten. If False and the file exists,
1052             the download is skipped.
1053         variables : list of str
1054             List of variable names to download (e.g., ['tasmax', 'tasmin']).
1055         year : int
1056             Year of the forecast initiation.
1057         initiation_month : int
1058             Month of the forecast initiation (1-12).
1059         data_format : str
1060             File format for the downloaded data ('grib' or 'netcdf').
1061         originating_centre : str
1062             Forecast data provider (e.g., 'dwd' for German Weather Service).
1063         system : str
1064             Model system identifier (e.g., '21').
1065         bounds_cds_order : list of float
1066             Geographical bounding box in CDS order: [north, west, south, east].
1067         leadtimes : list of int
1068             List of forecast lead times in hours.
1069
1070         Returns
1071         -----
1072         Path
1073             Path to the downloaded data file.
1074
1075         Notes
1076         -----
1077         The function uses the `download_data` method from the Copernicus interface module.
1078         The downloaded data is stored following the directory structure defined by the p
1079         """
1080         # Prepare download parameters
1081         download_params = {
1082             "data_format": data_format,
1083             "originating_centre": originating_centre,
1084             "area": bounds_cds_order,
1085             "system": system,
1086             "variable": variables,
1087             "month": initiation_month,
1088             "year": year,

```

```

1089         "day": "01",
1090         "leadtime_hour": leadtimes,
1091     }
1092
1093     # Perform download
1094     downloaded_file = download_data(
1095         "seasonal-original-single-levels",
1096         download_params,
1097         output_file_name,
1098         overwrite=overwrite,
1099     )
1100
1101     return downloaded_file
1102
1103
1104     @handle_overwriting
1105     def _process_data(output_file_name, overwrite, input_file_name, variables, data_format):
1106         """
1107         Process a downloaded forecast data file into daily NetCDF format.
1108
1109         This function reads the downloaded forecast data (in GRIB or NetCDF format),
1110         applies a temporal coarsening operation (aggregation over 4 time steps),
1111         and saves the resulting daily data as a NetCDF file. For each variable,
1112         daily mean, maximum, and minimum values are computed.
1113
1114         Parameters
1115         -----
1116         output_file_name : Path
1117             Path to save the processed NetCDF file.
1118         overwrite : bool
1119             If True, existing processed files will be overwritten. If False and the file
1120             the processing is skipped.
1121         input_file_name : Path
1122             Path to the input downloaded data file.
1123         variables : list of str
1124             List of short variable names to process (e.g., ['tasmax', 'tasmin']).
1125         data_format : str
1126             Format of the input file ('grib' or 'netcdf').
1127
1128         Returns
1129         -----
1130         Path
1131             Path to the saved processed NetCDF file.
1132
1133         Raises
1134         -----
1135         FileNotFoundError
1136             If the input file does not exist.
1137         Exception
1138             If an error occurs during data processing.
1139
1140         Notes
1141         -----

```

```

1141 -----
1142 The function performs a temporal aggregation by coarsening the data over 4 time :
1143 resulting in daily mean, maximum, and minimum values for each variable.
1144 The processed data is saved in NetCDF format and can be used for index calculation
1145 """
1146 try:
1147     with xr.open_dataset(
1148         input_file_name,
1149         engine="cfgrib" if data_format == "grib" else None,
1150     ) as ds:
1151         # Coarsen the data
1152         ds_mean = ds.coarsen(step=4, boundary="trim").mean()
1153         ds_max = ds.coarsen(step=4, boundary="trim").max()
1154         ds_min = ds.coarsen(step=4, boundary="trim").min()
1155
1156         # Create a new dataset combining mean, max, and min values
1157         combined_ds = xr.Dataset()
1158         for var in variables:
1159             combined_ds[f"{var}_mean"] = ds_mean[var]
1160             combined_ds[f"{var}_max"] = ds_max[var]
1161             combined_ds[f"{var}_min"] = ds_min[var]
1162
1163         # Save the combined dataset to NetCDF
1164         combined_ds.to_netcdf(str(output_file_name))
1165         LOGGER.info("Daily file saved to %s", output_file_name)
1166
1167         return output_file_name
1168
1169 except FileNotFoundError as error:
1170     raise FileNotFoundError(
1171         f"Input file {input_file_name} does not exist. Processing failed."
1172     ) from error
1173 except Exception as error:
1174     raise Exception(
1175         f"Error during processing for {input_file_name}: {error}"
1176     ) from error
1177
1178
1179 @handle_overwriting
1180 ✓ def _calculate_index(
1181     output_file_names,
1182     overwrite,
1183     input_file_name,
1184     index_metric,
1185     tr_threshold=20,
1186     hw_threshold=27,
1187     hw_min_duration=3,
1188     hw_max_gap=0,
1189 ):
1190     """
1191     Calculate and save climate indices based on the input data.
1192
1193     Parameters

```

```

1194     -----
1195     output_file_names : dict
1196         Dictionary containing paths for daily, monthly, and stats output files.
1197     overwrite : bool
1198         Whether to overwrite existing files.
1199     input_file_name : Path
1200         Path to the input file.
1201     index_metric : str
1202         Climate index to calculate (e.g., 'HW', 'TR').
1203     threshold : float, optional
1204         Threshold for the index calculation (specific to the index type).
1205     min_duration : int, optional
1206         Minimum duration for events (specific to the index type).
1207     max_gap : int, optional
1208         Maximum gap allowed between events (specific to the index type).
1209     tr_threshold : float, optional
1210         Threshold for tropical nights (specific to the 'TR' index).
1211
1212     Returns
1213     -----
1214     dict
1215         Paths to the saved index files.
1216     """
1217     # Define output paths
1218     daily_output_path = output_file_names["daily"]
1219     monthly_output_path = output_file_names["monthly"]
1220     stats_output_path = output_file_names["stats"]
1221
1222     ds_daily, ds_monthly, ds_stats = seasonal_statistics.calculate_heat_indices_metric(
1223         input_file_name,
1224         index_metric,
1225         tr_threshold=tr_threshold,
1226         hw_threshold=hw_threshold,
1227         hw_min_duration=hw_min_duration,
1228         hw_max_gap=hw_max_gap,
1229     )
1230
1231     # Save outputs
1232     if ds_daily is not None:
1233         ds_daily.to_netcdf(daily_output_path)
1234         LOGGER.info("Saved daily index to %s", daily_output_path)
1235     if ds_monthly is not None:
1236         ds_monthly.to_netcdf(monthly_output_path)
1237         LOGGER.info("Saved monthly index to %s", monthly_output_path)
1238     if ds_stats is not None:
1239         ds_stats.to_netcdf(stats_output_path)
1240         LOGGER.info("Saved stats index to %s", stats_output_path)
1241
1242     return {
1243         "daily": daily_output_path,
1244         "monthly": monthly_output_path,
1245         "stats": stats_output_path,
1246     }

```



```

1246
1247
1248
1249     @handle_overwriting
1250     def _convert_to_hazard(output_file_name, overwrite, input_file_name, index_metric):
1251         """
1252         Convert a climate index file to a CLIMADA Hazard object and save it as HDF5.
1253
1254         This function reads a processed climate index NetCDF file, converts it to a
1255         CLIMADA Hazard object, and saves it in HDF5 format. The function supports
1256         ensemble members and concatenates them into a single Hazard object.
1257
1258         Parameters
1259         -----
1260         output_file_name : Path
1261             Path to save the generated Hazard HDF5 file.
1262         overwrite : bool
1263             If True, existing hazard files will be overwritten. If False and the file exists,
1264             the conversion is skipped.
1265         input_file_name : Path
1266             Path to the input NetCDF file containing the calculated climate index.
1267         index_metric : str
1268             Climate index metric used for hazard creation (e.g., 'HW', 'TR', 'Tmax').
1269
1270         Returns
1271         -----
1272         Path
1273             Path to the saved Hazard HDF5 file.
1274
1275         Raises
1276         -----
1277         KeyError
1278             If required variables (e.g., 'step' or index variable) are missing in the dataset.
1279         Exception
1280             If the hazard conversion process fails.
1281
1282         Notes
1283         -----
1284         - The function uses `Hazard.from_xarray_raster()` to create Hazard objects
1285           from the input dataset.
1286         - If multiple ensemble members are present, individual Hazard objects are
1287           created for each member and concatenated.
1288         - The function determines the intensity unit based on the selected index:
1289           - '%' for relative humidity (RH)
1290           - 'days' for duration indices (e.g., 'HW', 'TR', 'TX30')
1291           - '°C' for temperature indices
1292         """
1293         try:
1294             with xr.open_dataset(str(input_file_name)) as ds:
1295                 if "step" not in ds.variables:
1296                     raise KeyError(
1297                         f"Missing 'step' variable in dataset for {input_file_name}."
1298                     )

```

```

1299
1300     ds["step"] = xr.DataArray(
1301         [f"{date}-01" for date in ds["step"].values],
1302         dims=["step"],
1303     )
1304     ds["step"] = pd.to_datetime(ds["step"].values)
1305
1306     ensemble_members = ds.get("number", [0]).values
1307     hazards = []
1308
1309     # Determine intensity unit and variable
1310     intensity_unit = (
1311         "%"
1312         if index_metric == "RH"
1313         else "days" if index_metric in ["TR", "TX30", "HW"] else "°C"
1314     )
1315     intensity_variable = index_metric
1316
1317     if intensity_variable not in ds.variables:
1318         raise KeyError(
1319             f"No variable named '{intensity_variable}' in the dataset. "
1320             f"Available variables: {list(ds.variables)}"
1321         )
1322
1323     # Create Hazard objects
1324     for member in ensemble_members:
1325         ds_subset = ds.sel(number=member) if "number" in ds.dims else ds
1326         hazard = Hazard.from_xarray_raster(
1327             data=ds_subset,
1328             hazard_type=index_metric,
1329             intensity_unit=intensity_unit,
1330             intensity=intensity_variable,
1331             coordinate_vars={
1332                 "event": "step",
1333                 "longitude": "longitude",
1334                 "latitude": "latitude",
1335             },
1336         )
1337         hazard.event_name = [
1338             f"member{member}" for _ in range(len(hazard.event_name))
1339         ]
1340         hazards.append(hazard)
1341
1342     hazard = Hazard.concat(hazards)
1343     hazard.check()
1344     hazard.write_hdf5(str(output_file_name))
1345
1346     LOGGER.info("Hazard file saved to %s.", output_file_name)
1347     return output_file_name
1348
1349 except Exception as error:
1350     raise Exception(
1351         f"Failed to convert {input file name} to hazard: {error}"

```

