helper.py

```python
import collections
import functools
import json
import operator
import os.path
import warnings
from functools import reduce
from typing import List, Dict

import dace
import numpy as np

"""
    This file contains many helper methods that are being re-used in multiple classes and do not specifically belong to
    a class.
"""


def deprecated(func):
    """
    This is a decorator which can be used to mark functions as deprecated. It will result in a warning being emitted
    when the function is used.
    """
    @functools.wraps(func)
    def new_func(*args, **kwargs):
        warnings.simplefilter('always', DeprecationWarning)  # turn off filter
        warnings.warn(
            "Call to deprecated function {}.".format(func.__name__),
            category=DeprecationWarning,
            stacklevel=2)
        warnings.simplefilter('default', DeprecationWarning)  # reset filter
        return func(*args, **kwargs)
    return new_func


def str_to_dtype(dtype_str: str) -> dace.types.typeclass:
    """
    Conversion from the data type name (string) to its type defined in dace.
    :param dtype_str: string data type
    :return: dace data type
    """
    if not isinstance(dtype_str, str):  # type check
        raise TypeError("Expected string, got: " + type(dtype_str).__name__)
    try:
        return getattr(dace.types, dtype_str)  # match type
    except AttributeError:
        pass
    raise AttributeError("Unsupported data type: " + dtype_str)  # missmatch
```

helper.py

```
def parse_json(config_path: str) -> Dict:
    """
    Read input file from disk and parse it.
    :param config_path: path to the file
    :return: parsed file
    """
    # check file exists
    if not os.path.isfile(config_path):
        relative = os.path.join(
            os.path.dirname(os.path.realpath(__file__)), config_path)
        if not os.path.isfile(relative):
            raise RuntimeError("file {} does not exists.".format(config_path))
        config_path = relative
    # open file in with-clause, to ensure proper file closing even in the event of an exception
    with open(config_path, "r") as file_handle:
        # try to parse it
        config = json.loads(file_handle.read())  # type: dict
    # Save the path to the config
    config["path"] = os.path.dirname(os.path.abspath(config_path))
    def walk(d):  # replace string data type in config
        for key, val in d.items():
            if isinstance(val, dict):
                walk(val)
            else:
                if key == "data_type":
                    d[key] = str_to_dtype(val)
    walk(config)
    # return dict
    return config


def max_dict_entry_key(dict1: Dict[str, List[int]]) -> str:
    """
    Get key of largest value entry out of the input dictionary.
    :param dict1: a dictionary with keys as names and values as buffer sizes
    :return: key of buffer entry with maximum size
    """
    # check type
    if not isinstance(dict1, dict):
        raise Exception("dict1 should be of type {}, but is of type {}".format(
            type(dict), type(dict1)))
    # extract max value entry
    return max(dict1, key=dict1.get)


def list_add_cwise(list1: List,
                   list2: List) -> List:
    """
    Merge two lists by component-wise addition.
    :param list1: input list: summand
    :param list2: input list: summand
    :return: merged list
    """
    # check type
    if not isinstance(list1, list):
        raise Exception("list1 should be of type {}, but is of type {}".format(
            type(list), type(list1)))
    if not isinstance(list2, list):
        raise Exception("list2 should be of type {}, but is of type {}".format(
            type(list), type(list2)))
    # do map lambda operation over both lists
    return list(map(lambda x, y: x + y, list1, list2))


def list_subtract_cwise(list1: List,
                        list2: List) -> List:
    """
    Merge two lists by component-wise subtraction.
    :param list1: input list: minuend
    :param list2: input list: subtrahend
    :return: merged list
    """
    # check type
    if not isinstance(list1, list):
        raise Exception("list1 should be of type {}, but is of type {}".format(
            type(list), type(list1)))
    if not isinstance(list2, list):
        raise Exception("list2 should be of type {}, but is of type {}".format(
            type(list), type(list2)))
    # do map lambda operation over both lists
    return list(map(lambda x, y: x - y, list1, list2))


def dim_to_abs_val(input: List[int],
                   dimensions: List[int]) -> int:
    """
    Computes scalar number out of independent dimension unit.
    :param input: vector to evaluate
    :param dimensions: vector of global array dimensions
    :return: scalar value
    """
    # dim [X, Y, Z], size [a, b, c] -> 1*c + X*(b + Y*a) = [a, b, c] * transpose([Z*Y, Z, 1])
    vec = [
        reduce(operator.mul, dimensions[i + 1:], 1)
        for i in range(len(dimensions))
    ]
    return reduce(operator.add, map(operator.mul, input, vec))  # inner product
```

helper.py

```
def load_array(source_config: Dict, search_path=None):
    """
    Load array from file or list into numpy array.
    :param source_config: External data input file config.
    :param search_path: Path to search as relative paths.
    :return: Data stored in a numpy array.
    """
    # get path to either source file or direct to the embedded array
    data = source_config["data"]
    if isinstance(data, str):  # source file
        path = data
        if not os.path.isfile(path):
            if search_path is not None:
                path = os.path.join(search_path, path)
            if not os.path.isfile(path):
                raise RuntimeError("File {} does not exists.".format(data))
        if path.endswith(".csv"):
            return np.genfromtxt(path, float, delimiter=',')
        elif path.endswith(".dat"):
            return np.fromfile(path, float)
        else:
            raise ValueError("Invalid file type: " + path)
    elif isinstance(data, np.ndarray):  # embedded array: already numpy array
        return data
    else:
        return np.array(data, dtype=source_config["data_type"].type) # embedded array: collection item -> convert to
        # np array


def load_input_arrays(program: Dict) -> Dict:
    """
    Loads input arrays for the passed program into memory.
    :param program: Program tree as generated by parse_json.
    :return: Dictionary of input names to input data as numpy arrays.
    """
    # add all input arrays to the dict
    input_arrays = dict()
    for arr_name, source in program["inputs"].items():
        input_arrays[arr_name] = load_array(source, program["path"])
    return input_arrays


def save_array(array, path):
    """
    Saves array to a binary file.
    :param array: numpy array to save.
    :param path: Path to binary file where data will be saved.
    """
    array.tofile(path)


def save_output_arrays(outputs: Dict, output_dir=str()):
    """
    Saves output arrays to individual files.
    :param outputs: Dictionary of array names to numpy arrays.
    :param output_dir: Folder to store files in.
    """
    # store all arrays in the output directory path
    for arr_name, arr_data in outputs.items():
        path = os.path.join(output_dir, arr_name + ".dat")
        save_array(arr_data, path)


def arrays_are_equal(reference, result, tolerance=1e-3):
    """
    Check if two arrays are equal within a tolerance.
    :param reference: numpy array or path to file.
    :param result: numpy array or path to file.
    :param tolerance: Maximum relative (fractional) difference between arrays.
    """
    if not isinstance(reference, np.ndarray):
        reference = load_array(reference)
    if not isinstance(result, np.ndarray):
        result = load_array(result)
    # tolerate zeroes by adding epsilon to the divisor
    relative_diff = (np.abs(reference - result) / (np.maximum.reduce([reference, result]) + np.finfo(np.float64).eps))
    return np.all(relative_diff <= tolerance)


def unique(iterable):
    """
    Removes duplicates in the passed iterable.
    :param iterable: iterable with potential duplicates
    :return iterable without duplicates
    """
    try:
        return type(iterable)([i for i in sorted(set(iterable), key=lambda x: iterable.index(x))])
    except TypeError:
        return type(iterable)(collections.OrderedDict(
            zip(map(str, iterable), iterable)).values())


def convert_3d_to_1d(dimensions: List[int], index: List[int]) -> int:
    """
    Convert the size of form [a, b, c] to absolute values according to the global dimensions of the problem.
    :param dimensions: problem size (e.g. size of the input arrays)
    :param index: 3d value of the size
    :return the 1d value of the size
    """
    # convert [i, j, k] to flat 1D array index using the given dimensions [dimX, dimY, dimZ]
    # index = i*dimY*dimZ + j*dimZ + k = (i*dimY + j)*dimZ + k
    if not index:
        return 0  # empty list
    return dim_to_abs_val(index, dimensions)


if __name__ == "__main__":
    """
        Basic helper function test. Comprehensive testing is implemented in 'testing.py'.
    """
    example_list = [[1, 2, 2], [1, 2, 3], [3, 2, 1], [2, 3, 1]]
    print("properties of list {}:\nmin: {}\nmax: {}\n".format(
        example_list, min(example_list), max(example_list)))

    example_dict = {
        "small": [0, 10, 10],
        "very small": [0, 1, 0],
        "extra large": [12, 1, 2],
        "large": [10, 10, 10]
    }
    print("max value entry key of dict {} is:\n\'{}\'".format(
        example_dict, max_dict_entry_key(example_dict)))
```