```
# from __future__ import annotations  # support return type of its own class
import ast
from abc import ABCMeta, abstractmethod
from enum import Enum
from typing import List, Dict

import dace.types

from bounded_queue import BoundedQueue


class BoundaryCondition(Enum):
    """
        The BoundaryCondition Enumeration works as an adapter between the input string representation and the
        programmatically more useful enumeration. It defines the strategy used for out-of-bound stencil accesses on
        the data arrays we iterate over.
    """

    CONSTANT = 1  # use a fixed (static) value for all out-of-bound accesses
    COPY = 2  # copy the last within-bound-value for out-of-bound accesses

    @staticmethod
    def to_bc(text: str):  # -> BoundaryCondition:
        if text == "const":
            return BoundaryCondition.CONSTANT
        elif text == "copy":
            return BoundaryCondition.COPY
        else:
            raise Exception("{} is not a valid boundary condition string".format(text))


class BaseKernelNodeClass:
    """
        The BaseKernelClass provides all the basic fields and functionality for its subclasses which are the Input,
        Kernel and Output classes. These are nodes of of the KernelChainGraph.
    """

    __metaclass__ = ABCMeta

    def __init__(self, name: str,
                 data_queue: BoundedQueue,
                 data_type: dace.types.typeclass,
                 verbose: bool = False) -> None:
        """
        Create new BaseKernelNodeClass with given initialization parameters.
        :param name: name of the node
        :param data_queue: queue containing the input (Input) or final output (Output) data
        :param data_type: set whether or not the buffer is swapped out (might get overridden by the optimizer)
        :param verbose: flag for console output logging
```

```
        """
        # save params
        self.name: str = name
        self.data_queue: BoundedQueue = data_queue
        self.data_type = data_type
        if not isinstance(data_type, dace.types.typeclass):  # check type of input
            raise TypeError("Expected dace.types.typeclass, got: " + type(data_type).__name__)
        self.verbose = verbose
        # define basic node structures
        self.input_paths: Dict[str, List] = dict()  # contains all paths to the source arrays
        self.inputs: Dict[str, Dict] = dict()  # contains all predecessors
        self.outputs: Dict[str, BoundedQueue] = dict()  # contains all successors
        self.delay_buffer: Dict[str, BoundedQueue] = dict()  # contains the delay buffers for all inputs
        self.program_counter = 0  # progress program counter for simulation

    def generate_label(self) -> str:  # wrapper for customizations
        """
        Base class basic implementation of the generate_label method.
        :returns generated label
        """
        return self.name


class BaseOperationNodeClass:
    """
        The BaseOperationNodeClass class provides all the basic fields and methods for its subclasses (Num,
        Subscript,..). These are the nodes of the ComputeGraph .
    """

    __metaclass__ = ABCMeta

    def __init__(self,
                 ast_node: ast,
                 number: int,
                 verbose: bool = False) -> None:
        """
        Create new BaseOperationNodeClass with given initialization parameters.
        :param ast_node: abstract syntax tree (python) entry
        :param number: node number (tree numbering)
        :param verbose: flag for console output logging
        """
        # save params
        self.number: int = number
        self.name: str = self.generate_name(ast_node)
        self.verbose = verbose
        # set initial latency to a value distinguishable from correct values
        self.latency: int = -1

    @abstractmethod
```

base_node_class.py

```
def generate_name(self,
                  ast_node: ast) -> str:  # every subclass must implement this
    """
    Base class basic implementation of the generate_label method.
    :returns generated label
    """
    return str(self.name)

def generate_label(self) -> str:  # subclass can, if necessary, override the default implementation
    """
    Generates the node label.
    :returns generated label
    """
    return str(self.name)
```