```python
from __future__ import annotations

from typing import Union
from dataclasses import dataclass
from collections.abc import Sequence
from functools import cached_property

import numpy as np

from gpaw.response.kpoints import KPointDomainGenerator


@dataclass
class QSymmetries(Sequence):
    """Symmetry operations for a given q-point.

    We operate with several different symmetry indices:
      * u: indices the unitary symmetries of the system. Length is nU.
      * S: extended symmetry index. In addition to the unitary symmetries
           (first nU indices) it includes also symmetries generated by a
           unitary symmetry transformation *followed* by a time-reversal.
           Length is 2 * nU.
      * s: reduced symmetry index. Includes all the "S-symmetries" which map
           the q-point in question onto itself (up to a reciprocal lattice
           vector). May be reduced further, if some of the symmetries have been
           disabled. Length is q-dependent and depends on user input.
    """
    q_c: np.ndarray
    U_ucc: np.ndarray  # unitary symmetry transformations
    S_s: np.ndarray  # extended symmetry index for each q-symmetry
    shift_sc: np.ndarray  # reciprocal lattice shifts, G = (T)Uq - q

    def __post_init__(self):
        self.nU = len(self.U_ucc)

    def __len__(self):
        return len(self.S_s)

    def __getitem__(self, s):
        return self.U_scc[s], self.sign_s[s], self.shift_sc[s]

    def unioperator(self, S):
        return self.U_ucc[S % self.nU]

    def timereversal(self, S):
        """Does the extended index S involve a time-reversal symmetry?"""
        return bool(S // self.nU)

    def sign(self, S):
        """Flip the sign under time-reversal."""
        if self.timereversal(S):
            return -1
        return 1

    @cached_property
    def U_scc(self):
        return np.array([self.unioperator(S) for S in self.S_s])

    @cached_property
    def sign_s(self):
        return np.array([self.sign(S) for S in self.S_s])

    @cached_property
    def ndirect(self):
        """Number of direct symmetries."""
        return sum(np.array(self.S_s) < self.nU)

    @property
    def nindirect(self):
```

```python
        """Number of indirect symmetries."""
        return len(self) - self.ndirect

    def description(self) -> str:
        """Return string description of symmetry operations."""
        isl = ['\n']
        nx = 6  # You are not allowed to use non-symmorphic syms (value 3)
        y = 0
        for y in range((len(self) + nx - 1) // nx):
            for c in range(3):
                tisl = []
                for x in range(nx):
                    s = x + y * nx
                    if s == len(self):
                        break
                    U_cc, sign, _ = self[s]
                    op_c = sign * U_cc[c]
                    tisl.append(f'  ({op_c[0]:2d} {op_c[1]:2d} {op_c[2]:2d})')
                tisl.append('\n')
                isl.append(''.join(tisl))
            isl.append('\n')
        return ''.join(isl[:-1])


QSymmetryInput = Union['QSymmetryAnalyzer', dict, bool]


@dataclass
class QSymmetryAnalyzer:
    """Identifies symmetries of the k-grid, under which q is invariant.

    Parameters
    ----------
    point_group : bool
        Use point group symmetry.
    time_reversal : bool
        Use time-reversal symmetry (if applicable).
    """
    point_group: bool = True
    time_reversal: bool = True

    @staticmethod
    def from_input(qsymmetry: QSymmetryInput) -> QSymmetryAnalyzer:
        if not isinstance(qsymmetry, QSymmetryAnalyzer):
            if isinstance(qsymmetry, dict):
                qsymmetry = QSymmetryAnalyzer(**qsymmetry)
            else:
                qsymmetry = QSymmetryAnalyzer(
                    point_group=qsymmetry, time_reversal=qsymmetry)
        return qsymmetry

    @property
    def disabled(self):
        return not (self.point_group or self.time_reversal)

    @property
    def disabled_symmetry_info(self):
        if self.disabled:
            txt = ''
        elif not self.point_group:
            txt = 'point-group '
        elif not self.time_reversal:
            txt = 'time-reversal '
        else:
            return ''
        txt += 'symmetry has been manually disabled'
        return txt

    def analysis_info(self, symmetries):
```

```python
        dsinfo = self.disabled_symmetry_info
        return '\n'.join([
            '',
            f'Symmetries of q_c{f" ({dsinfo})" if len(dsinfo) else ""}:',
            f'    Direct symmetries (Uq -> q): {symmetries.ndirect}',
            f'    Indirect symmetries (TUq -> q): {symmetries.nindirect}',
            f'In total {len(symmetries)} allowed symmetries.',
            symmetries.description()])

    def analyze(self, q_c, kpoints, context):
        """Analyze symmetries and set up KPointDomainGenerator."""
        symmetries = self.analyze_symmetries(q_c, kpoints.kd)
        generator = KPointDomainGenerator(symmetries, kpoints)
        context.print(self.analysis_info(symmetries))
        context.print(generator.get_infostring())
        return symmetries, generator

    def analyze_symmetries(self, q_c, kd):
        r"""Determine allowed symmetries.

        An direct symmetry U must fulfill::

          U \mathbf{q} = q + \Delta

        Under time-reversal (indirect) it must fulfill::

          -U \mathbf{q} = q + \Delta

        where :math:`\Delta` is a reciprocal lattice vector.
        """
        # Map q-point for each unitary symmetry
        U_ucc = kd.symmetry.op_scc  # here s is the unitary symmetry index
        Uq_uc = np.dot(U_ucc, q_c)

        # Direct and indirect -> global symmetries
        nU = len(U_ucc)
        nS = 2 * nU
        shift_Sc = np.zeros((nS, 3), int)
        is_qsymmetry_S = np.zeros(nS, bool)

        # Identify direct symmetries
        # Check whether U q - q is integer (reciprocal lattice vector)
        dshift_uc = Uq_uc - q_c[np.newaxis]
        is_direct_symm_u = (dshift_uc == dshift_uc.round()).all(axis=1)
        is_qsymmetry_S[:nU][is_direct_symm_u] = True
        shift_Sc[:nU] = dshift_uc

        # Identify indirect symmetries
        # Check whether -U q - q is integer (reciprocal lattice vector)
        idshift_uc = -Uq_uc - q_c
        is_indirect_symm_u = (idshift_uc == idshift_uc.round()).all(axis=1)
        is_qsymmetry_S[nU:][is_indirect_symm_u] = True
        shift_Sc[nU:] = idshift_uc

        # The indices of the allowed symmetries
        S_s = is_qsymmetry_S.nonzero()[0]

        # Set up symmetry filters
        def is_not_point_group(S):
            return (U_ucc[S % nU] == np.eye(3)).all()

        def is_not_time_reversal(S):
            return not bool(S // nU)

        def is_not_non_symmorphic(S):
            return not bool(kd.symmetry.ft_sc[S % nU].any())

        # Filter out point-group symmetry, if disabled
        if not self.point_group:
```

```
    S_s = list(filter(is_not_point_group, S_s))

# Filter out time-reversal, if inapplicable or disabled
if not kd.symmetry.time_reversal or \
   kd.symmetry.has_inversion or \
   not self.time_reversal:
    S_s = list(filter(is_not_time_reversal, S_s))

# We always filter out non-symmorphic symmetries
S_s = list(filter(is_not_non_symmorphic, S_s))

return QSymmetries(q_c, U_ucc, S_s, shift_Sc[S_s])
```