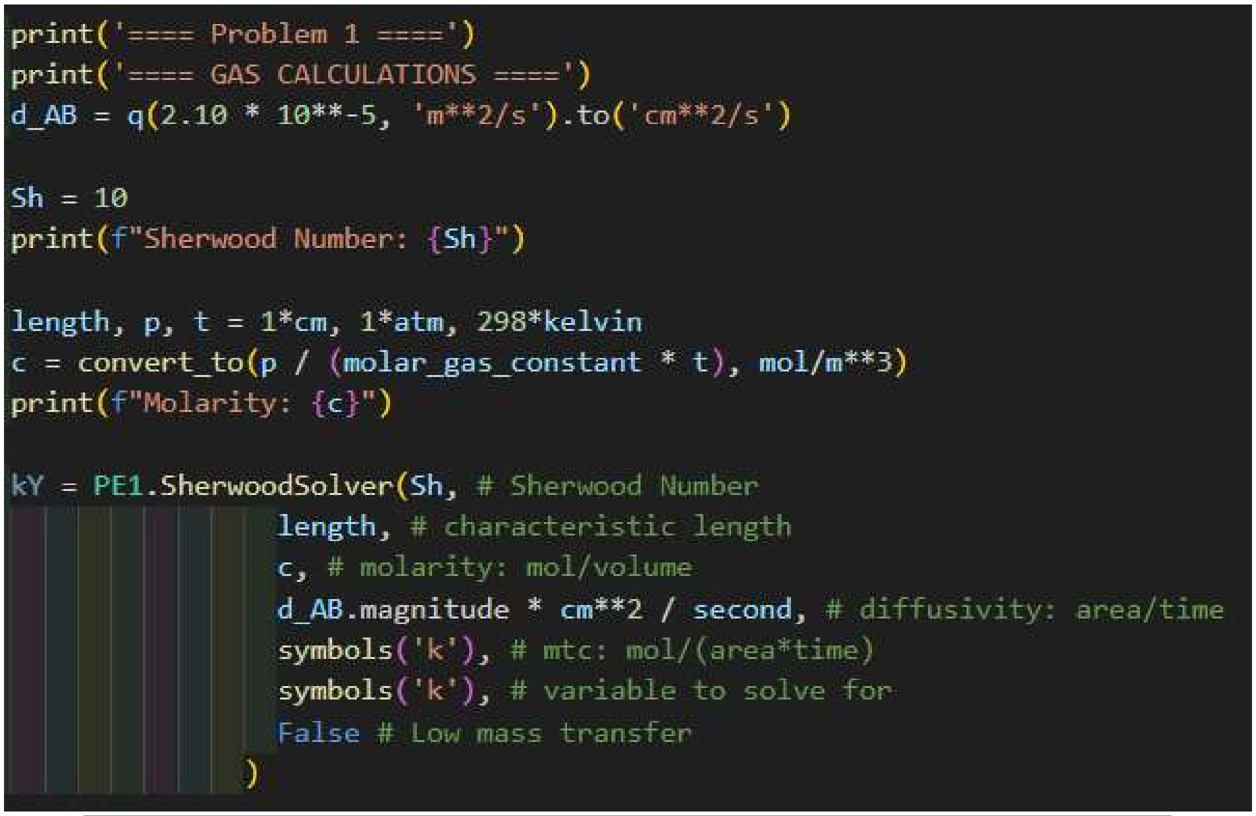
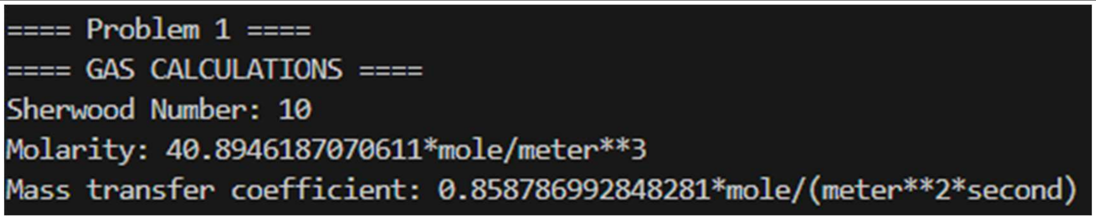
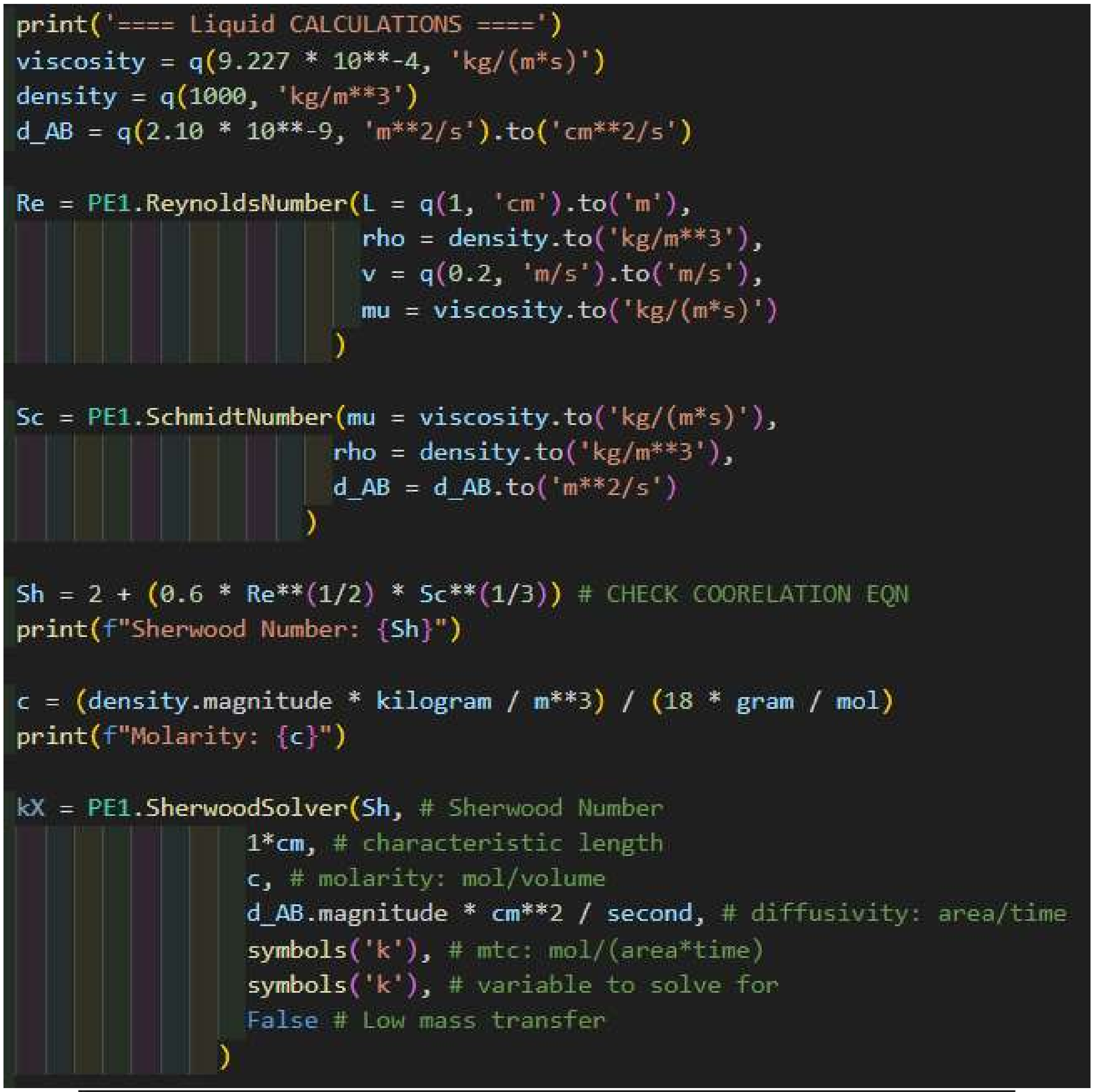


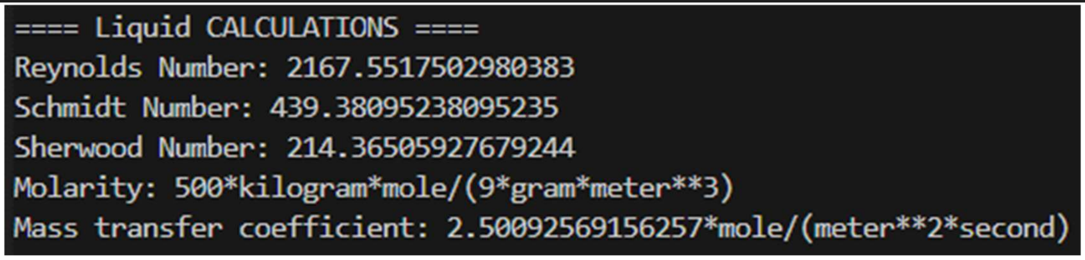
```python
print('==== Problem 1 ====')
print('==== GAS CALCULATIONS ====')
d_AB = q(2.10 * 10**-5, 'm**2/s').to('cm**2/s')

Sh = 10
print(f"Sherwood Number: {Sh}")

length, p, t = 1*cm, 1*atm, 298*kelvin
c = convert_to(p / (molar_gas_constant * t), mol/m**3)
print(f"Molarity: {c}")

kY = PE1.SherwoodSolver(Sh, # Sherwood Number
                length, # characteristic length
                c, # molarity: mol/volume
                d_AB.magnitude * cm**2 / second, # diffusivity: area/time
                symbols('k'), # mtc: mol/(area*time)
                symbols('k'), # variable to solve for
                False # Low mass transfer
               )
```



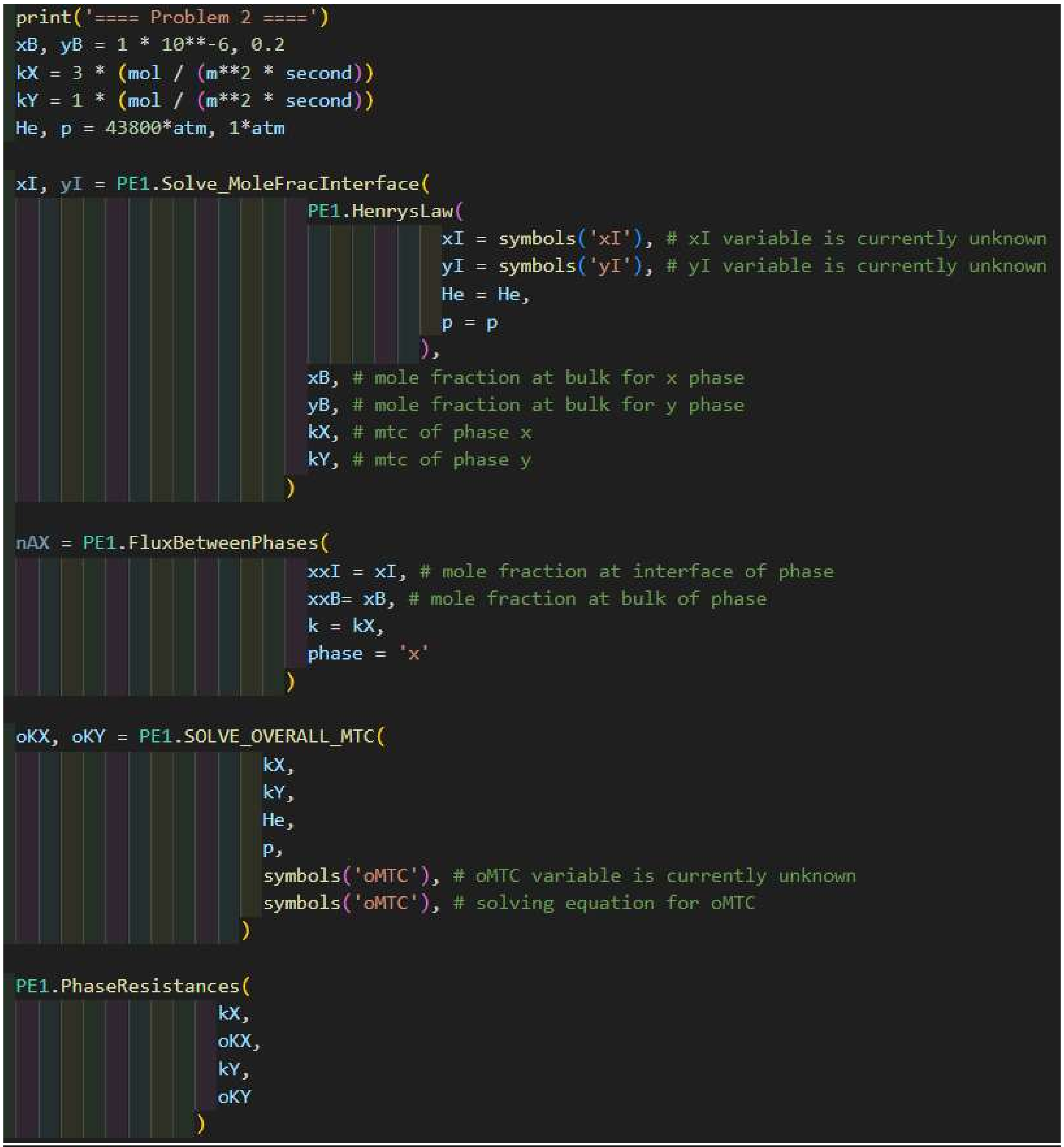
```
==== Problem 1 ====
==== GAS CALCULATIONS ====
Sherwood Number: 10
Molarity: 40.8946187070611*mole/meter**3
Mass transfer coefficient: 0.858786992848281*mole/(meter**2*second)
```



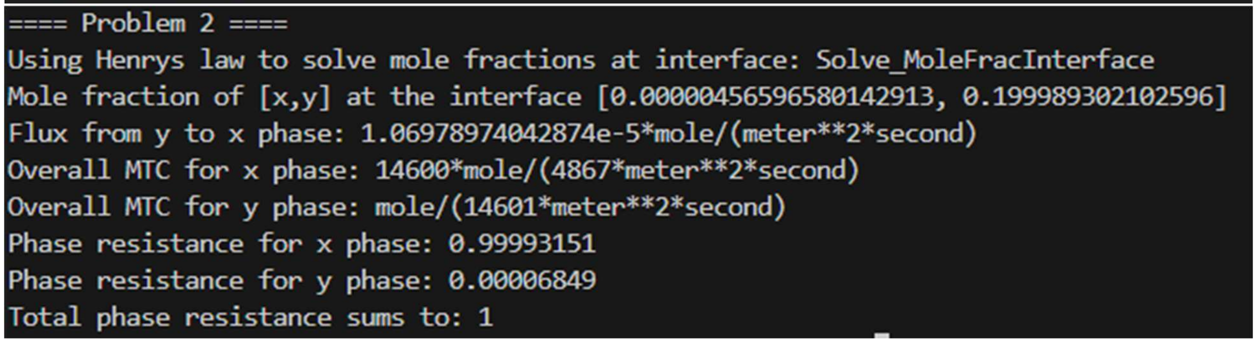
```python
print('==== Liquid CALCULATIONS ====')
viscosity = q(9.227 * 10**-4, 'kg/(m*s)')
density = q(1000, 'kg/m**3')
d_AB = q(2.10 * 10**-9, 'm**2/s').to('cm**2/s')

Re = PE1.ReynoldsNumber(L = q(1, 'cm').to('m'),
                        rho = density.to('kg/m**3'),
                        v = q(0.2, 'm/s').to('m/s'),
                        mu = viscosity.to('kg/(m*s)')
                       )

Sc = PE1.SchmidtNumber(mu = viscosity.to('kg/(m*s)'),
                       rho = density.to('kg/m**3'),
                       d_AB = d_AB.to('m**2/s')
                      )

Sh = 2 + (0.6 * Re**(1/2) * Sc**(1/3)) # CHECK COORELATION EQN
print(f"Sherwood Number: {Sh}")

c = (density.magnitude * kilogram / m**3) / (18 * gram / mol)
print(f"Molarity: {c}")

kX = PE1.SherwoodSolver(Sh, # Sherwood Number
                        1*cm, # characteristic length
                        c, # molarity: mol/volume
                        d_AB.magnitude * cm**2 / second, # diffusivity: area/time
                        symbols('k'), # mtc: mol/(area*time)
                        symbols('k'), # variable to solve for
                        False # Low mass transfer
                       )
```



```
==== Liquid CALCULATIONS ====
Reynolds Number: 2167.5517502980383
Schmidt Number: 439.38095238095235
Sherwood Number: 214.36505927679244
Molarity: 500*kilogram*mole/(9*gram*meter**3)
Mass transfer coefficient: 2.50092569156257*mole/(meter**2*second)
```

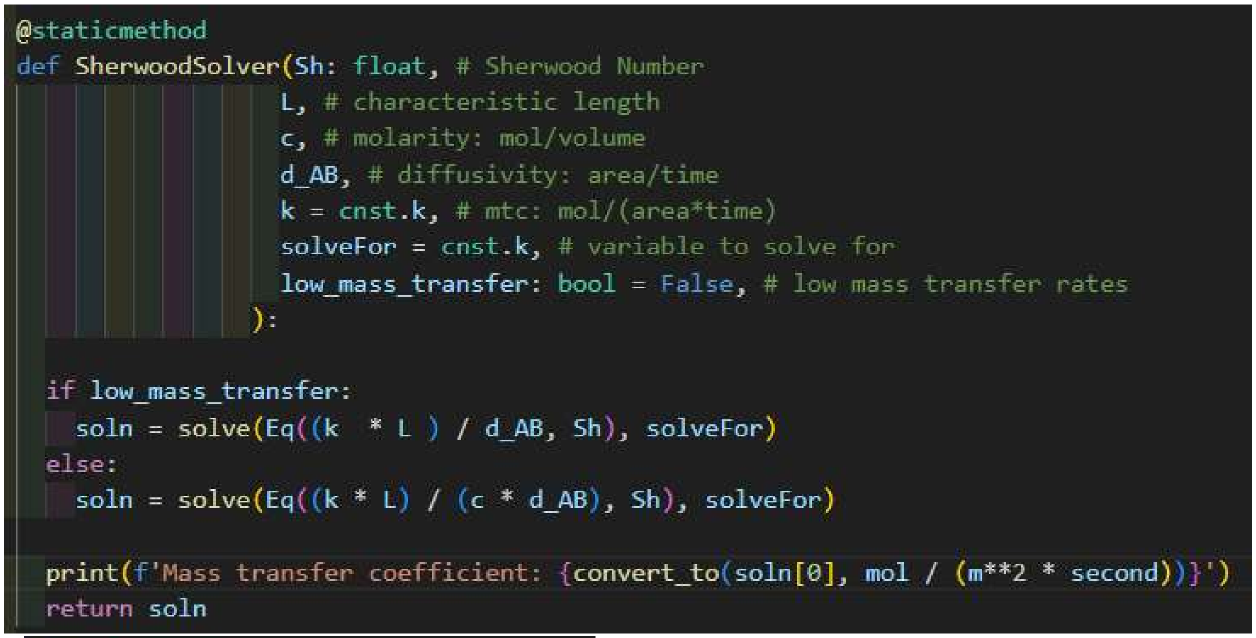


```
print('==== Problem 2 ====')
xB, yB = 1 * 10**-6, 0.2
kX = 3 * (mol / (m**2 * second))
kY = 1 * (mol / (m**2 * second))
He, p = 43800*atm, 1*atm

xI, yI = PE1.Solve_MoleFracInterface(
                                        PE1.HenrysLaw(
                                                    xI = symbols('xI'), # xI variable is currently unknown
                                                    yI = symbols('yI'), # yI variable is currently unknown
                                                    He = He,
                                                    p = p
                                                  ),
                                        xB, # mole fraction at bulk for x phase
                                        yB, # mole fraction at bulk for y phase
                                        kX, # mtc of phase x
                                        kY, # mtc of phase y
                                      )

nAX = PE1.FluxBetweenPhases(
                                        xxI = xI, # mole fraction at interface of phase
                                        xxB= xB, # mole fraction at bulk of phase
                                        k = kX,
                                        phase = 'x'
                                      )

oKX, oKY = PE1.SOLVE_OVERALL_MTC(
                                    kX,
                                    kY,
                                    He,
                                    p,
                                    symbols('oMTC'), # oMTC variable is currently unknown
                                    symbols('oMTC'), # solving equation for oMTC
                                  )

PE1.PhaseResistances(
                                kX,
                                oKX,
                                kY,
                                oKY
                              )
```

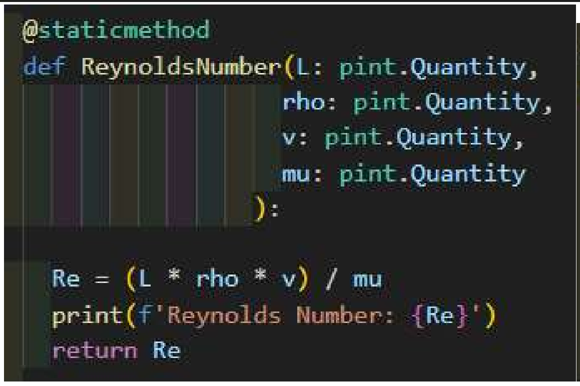


```
==== Problem 2 ====
Using Henrys law to solve mole fractions at interface: Solve_MoleFracInterface
Mole fraction of [x,y] at the interface [0.00000456596580142913, 0.199989302102596]
Flux from y to x phase: 1.06978974042874e-5*mole/(meter**2*second)
Overall MTC for x phase: 14600*mole/(4867*meter**2*second)
Overall MTC for y phase: mole/(14601*meter**2*second)
Phase resistance for x phase: 0.99993151
Phase resistance for y phase: 0.00006849
Total phase resistance sums to: 1
```

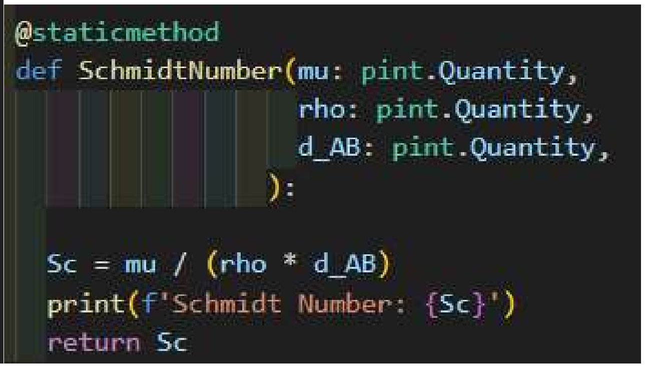
Base Methods:



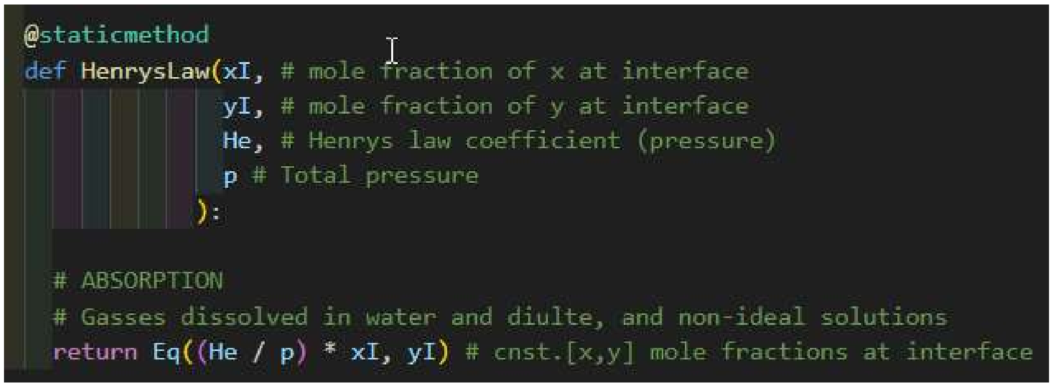
```python
@staticmethod
def SherwoodSolver(Sh: float, # Sherwood Number
                   L, # characteristic length
                   c, # molarity: mol/volume
                   d_AB, # diffusivity: area/time
                   k = cnst.k, # mtc: mol/(area*time)
                   solveFor = cnst.k, # variable to solve for
                   low_mass_transfer: bool = False, # low mass transfer rates
                  ):

    if low_mass_transfer:
        soln = solve(Eq((k  * L ) / d_AB, Sh), solveFor)
    else:
        soln = solve(Eq((k * L) / (c * d_AB), Sh), solveFor)

    print(f'Mass transfer coefficient: {convert_to(soln[0], mol / (m**2 * second))}')
    return soln
```



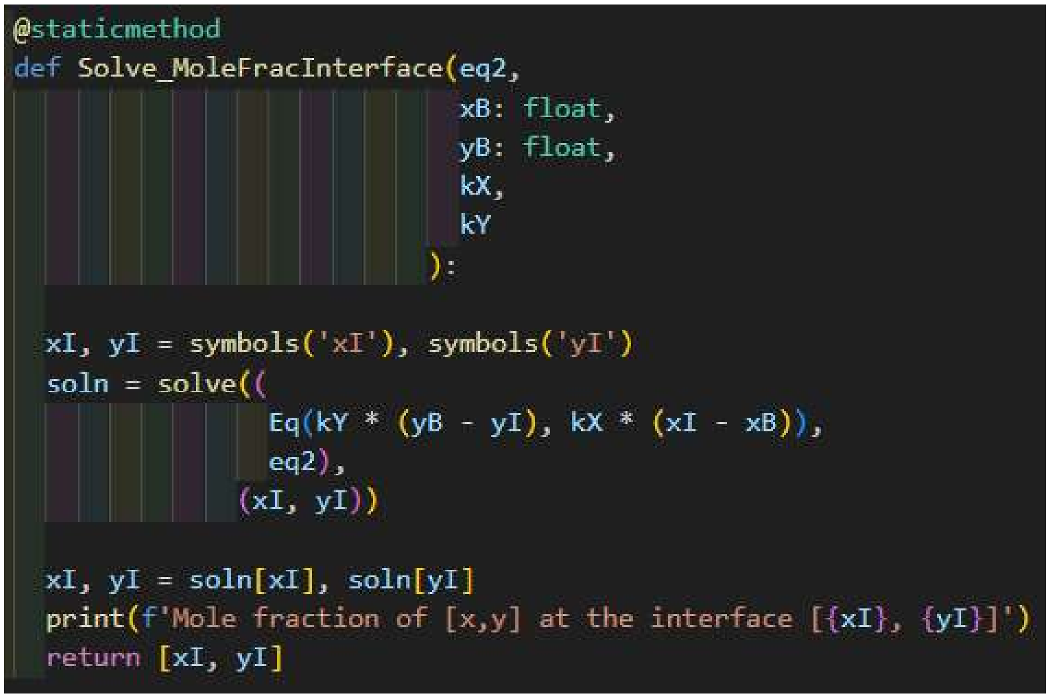
```python
@staticmethod
def ReynoldsNumber(L: pint.Quantity,
                   rho: pint.Quantity,
                   v: pint.Quantity,
                   mu: pint.Quantity
                  ):

    Re = (L * rho * v) / mu
    print(f'Reynolds Number: {Re}')
    return Re
```



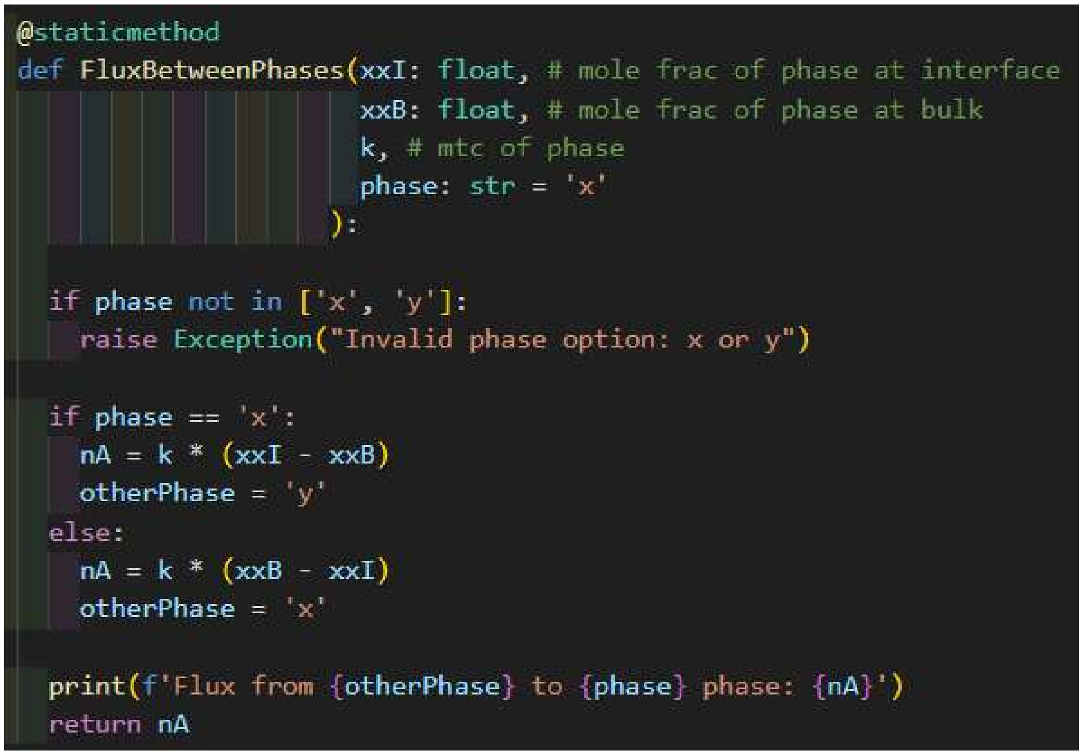
```python
@staticmethod
def SchmidtNumber(mu: pint.Quantity,
                  rho: pint.Quantity,
                  d_AB: pint.Quantity,
                 ):

    Sc = mu / (rho * d_AB)
    print(f'Schmidt Number: {Sc}')
    return Sc
```



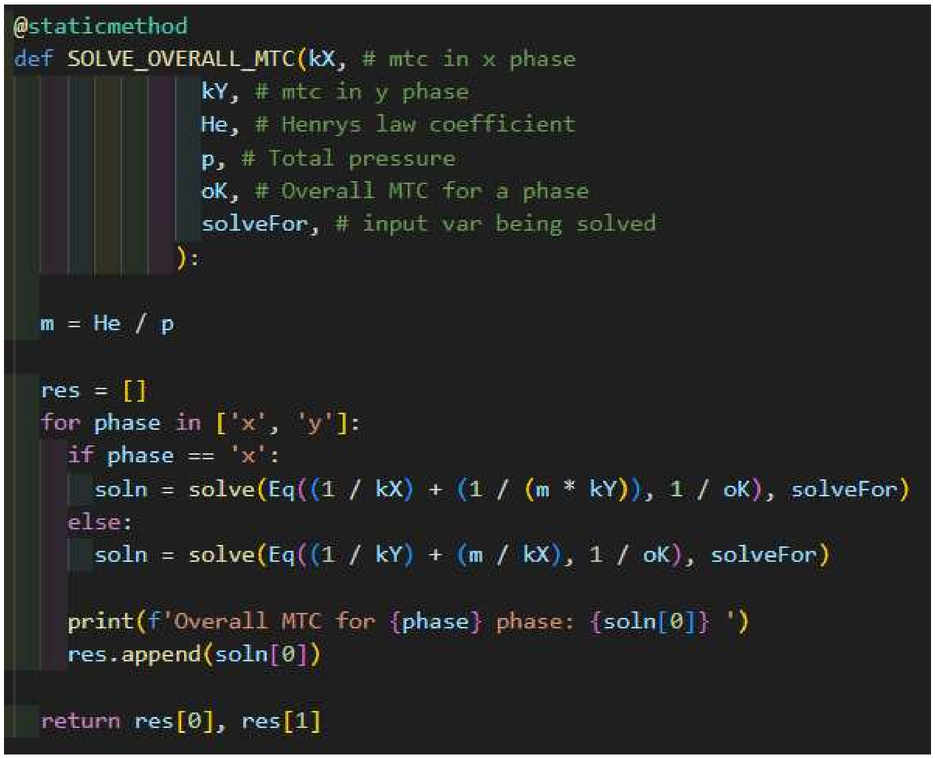
```python
@staticmethod
def HenrysLaw(xI, # mole fraction of x at interface
              yI, # mole fraction of y at interface
              He, # Henrys law coefficient (pressure)
              p # Total pressure
             ):

    # ABSORPTION
    # Gasses dissolved in water and diulte, and non-ideal solutions
    return Eq((He / p) * xI, yI) # cnst.[x,y] mole fractions at interface
```



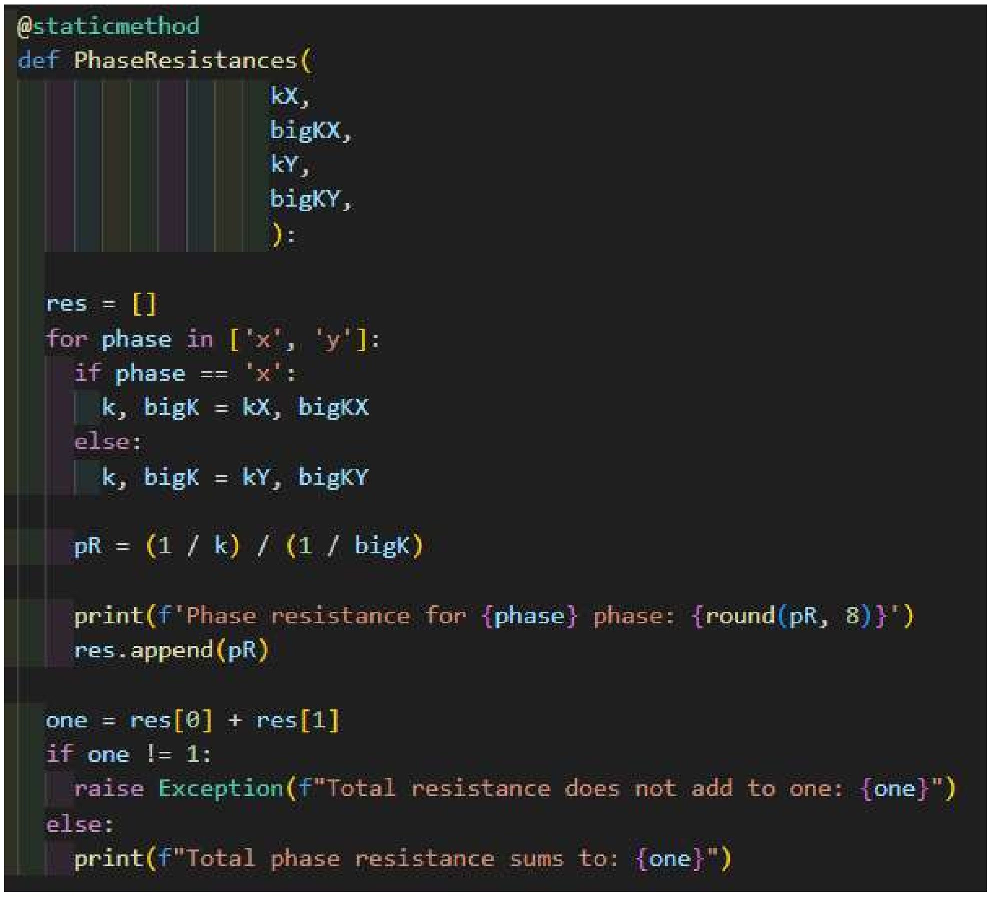
```python
@staticmethod
def Solve_MoleFracInterface(eq2,
                            xB: float,
                            yB: float,
                            kX,
                            kY
                           ):

    xI, yI = symbols('xI'), symbols('yI')
    soln = solve((
                  Eq(kY * (yB - yI), kX * (xI - xB)),
                  eq2),
                 (xI, yI))

    xI, yI = soln[xI], soln[yI]
    print(f'Mole fraction of [x,y] at the interface [{xI}, {yI}]')
    return [xI, yI]
```



```python
@staticmethod
def FluxBetweenPhases(xxI: float, # mole frac of phase at interface
                      xxB: float, # mole frac of phase at bulk
                      k, # mtc of phase
                      phase: str = 'x'
                     ):

    if phase not in ['x', 'y']:
        raise Exception("Invalid phase option: x or y")

    if phase == 'x':
        nA = k * (xxI - xxB)
        otherPhase = 'y'
    else:
        nA = k * (xxB - xxI)
        otherPhase = 'x'

    print(f'Flux from {otherPhase} to {phase} phase: {nA}')
    return nA
```



```python
@staticmethod
def SOLVE_OVERALL_MTC(kX, # mtc in x phase
                      kY, # mtc in y phase
                      He, # Henrys law coefficient
                      p, # Total pressure
                      oK, # Overall MTC for a phase
                      solveFor, # input var being solved
                     ):

    m = He / p

    res = []
    for phase in ['x', 'y']:
        if phase == 'x':
            soln = solve(Eq((1 / kX) + (1 / (m * kY)), 1 / oK), solveFor)
        else:
            soln = solve(Eq((1 / kY) + (m / kX), 1 / oK), solveFor)

        print(f'Overall MTC for {phase} phase: {soln[0]} ')
        res.append(soln[0])

    return res[0], res[1]
```



```python
@staticmethod
def PhaseResistances(
                     kX,
                     bigKX,
                     kY,
                     bigKY,
                    ):

    res = []
    for phase in ['x', 'y']:
        if phase == 'x':
            k, bigK = kX, bigKX
        else:
            k, bigK = kY, bigKY

        pR = (1 / k) / (1 / bigK)

        print(f'Phase resistance for {phase} phase: {round(pR, 8)}')
        res.append(pR)

    one = res[0] + res[1]
    if one != 1:
        raise Exception(f"Total resistance does not add to one: {one}")
    else:
        print(f"Total phase resistance sums to: {one}")
```