```python
from math import pi, sqrt


def get_user_input(problemType):
    """
    Handles user input radius type and related values.
    """
    moment = float(input("Moment (lb-in): ")) / 1000
    torque = float(input("Torque (lb-in): ")) / 1000

    if problemType == 4:
        diameter = 0.3
    else:
        diameter = float(input("Diameter (in): "))

    if problemType == 1 or problemType == 2 or problemType == 3:
        print("For... \n\t Sharp Radius: 1  \n\t Wide Radius: 2 \n\t Keyway: 3"
              " \n\t Retaining groove: 4")
        radiusType = int(input("Radius: "))

        if radiusType == 1:
            Kt = 2.7
            Kts = 2.2
            rootR = sqrt(diameter * 0.02)
        elif radiusType == 2:
            Kt = 1.7
            Kts = 1.5
            rootR = sqrt(diameter * 0.1)
        elif radiusType == 3:
            Kt = 2.14
            Kts = 3.0
            rootR = sqrt(diameter * 0.02)
        elif radiusType == 4:
            Kt = 5
            Kts = 3
            rootR = sqrt(0.01)
        else:
            Kt = Kts = rootR = 0
    else: Kt = Kts = rootR = 0
```

```python
    return moment, torque, diameter, Kt, Kts, rootR


def goodman_criteria(moment, torque, diameter, Kt, Kts, rootR):
    """
    Calculates the factor of safety using Goodman criteria for questions 1-10.
    """

    # Calculating stress concentration factors
    Kf = kf(rootR, Kt)
    Kfs = kfs(rootR, Kts)

    # Calculating endurance limit
    a, b = 2, -0.217
    Ka = a * Sut ** b
    Kb = 0.879 * diameter ** -0.107
    Kc = Kd = Ke = 1
    Se = Ka * Kb * Kc * Kd * Ke * sePrime

    # Calculating mean and alternating stress
    A = sqrt(4 * (Kf * moment) ** 2)
    B = sqrt(3 * (Kfs * torque) ** 2)

    Nf = ((pi * diameter ** 3) / 16) * ((A / Se) + (B / Sut)) ** -1
    return Nf


def vonmises_stress(moment, torque, diameter, Kt, Kts, rootR):
    """
    Calculates the safety factor against first cycle yielding using the full Von Mises.
    """

    # Calculating stress concentration factors
    Kf = kf(rootR, Kt)
    Kfs = kfs(rootR, Kts)

    # Setting up von mises calculation
    sigma = (32 * Kf * moment) / (pi * diameter ** 3)
    tau = (16 * Kfs * torque) / (pi * diameter ** 3)

    sigmaPrimeMax = sqrt((sigma ** 2) + (3 * tau ** 2))
```

```python
    return Sy / sigmaPrimeMax


def conservative_approximation(moment, torque, diameter, Kt, Kts, rootR):
    """
    Calculates the safety factor of first cycle yielding using the conservative approximation for
questions 16-20.
    """
    # Setting up conservative approximation
    sigmaPrimeA = (16 / (pi * diameter ** 3)) * sqrt(4 * (kf(rootR, Kt) * moment) ** 2)
    sigmaPrimeM = (16 / (pi * diameter ** 3)) * sqrt(3 * (kfs(rootR, Kts) * torque) ** 2)

    return Sy / (sigmaPrimeA + sigmaPrimeM)


def infinite_life(moment, torque, diameter):
    """
    Calculates the minimum diameter. This code was / is heavily bugged, and I am quickly losing
time. The required
    methods had to be copy and pasted. Goodman() and conservative() are the main methods that
*would* be called here.

    Stress concentration calculations are also copy and pasted to update with the new diameter

    """
    # Setting up an iterative approach to this problem set
    print("For... \n\t Sharp Radius: 1  \n\t Wide Radius: 2 \n\t Keyway: 3"
          " \n\t Retaining groove: 4")
    radiusType = int(input("Radius: "))

    while True:
        if radiusType == 1:
            Kt = 2.7
            Kts = 2.2
            rootR = sqrt(diameter * 0.02)
        elif radiusType == 2:
            Kt = 1.7
            Kts = 1.5
            rootR = sqrt(diameter * 0.1)
        elif radiusType == 3:
```

```
        Kt = 2.14
        Kts = 3.0
        rootR = sqrt(diameter * 0.02)
    elif radiusType == 4:
        Kt = 5
        Kts = 3
        rootR = sqrt(0.01)
    else:
        Kt = Kts = rootR = 0

    # Calculating sqrt(a)
    torsionalRootA = 0.190 - (2.51 * 10 ** -3) * Sut + (1.35 * 10 ** -5) * Sut ** 2 - (
            2.67 * 10 ** -8) * Sut ** 3

    # Calculating the notch sensitivity factor
    qTorsional = 1 / (1 + (torsionalRootA / rootR))
    Kfs = 1 + qTorsional * (Kts - 1)

    # Goodman approach
    a, b = 2, -0.217
    Ka = a * Sut ** b
    Kb = 0.879 * diameter ** -0.107
    Kc = Kd = Ke = 1
    Se = Ka * Kb * Kc * Kd * Ke * sePrime

    # Calculating sqrt(a)
    bendingRootA = 0.246 - (3.08 * 10 ** -3) * Sut + (1.51 * 10 ** -5) * Sut ** 2 - (2.67 * 10 ** -8)
* Sut ** 3

    # Calculating the notch sensitivity factor
    qBending = 1 / (1 + (bendingRootA / rootR))
    Kf = 1 + qBending * (Kt - 1)

    # Calculating sqrt(a)
    bendingRootA = 0.246 - (3.08 * 10 ** -3) * Sut + (1.51 * 10 ** -5) * Sut ** 2 - (2.67 * 10 ** -8)
* Sut ** 3

    # Calculating the notch sensitivity factor
    qBending = 1 / (1 + (bendingRootA / rootR))
    1 + qBending * (Kt - 1)
```

```
        # Calculating mean and alternating stress
        A = sqrt(4 * (Kf * moment) ** 2)
        B = sqrt(3 * (Kfs * torque) ** 2)

        # Calculating the goodman criteria
        goodman = ((pi * diameter ** 3) / 16) * ((A / Se) + (B / Sut)) ** -1

        # Conservative approach
        sigmaPrimeA = (16 / (pi * diameter ** 3)) * sqrt(4 * (Kf * moment) ** 2)
        sigmaPrimeM = (16 / (pi * diameter ** 3)) * sqrt(3 * (Kfs * torque) ** 2)
        conservative = Sy / (sigmaPrimeA + sigmaPrimeM)

        if goodman >= 1.5 and conservative >= 1.5:
            return diameter

        diameter += 0.00001


def kf(rootR, Kt):
    """
    Calculates the bending fatigue stress-concentration.
    """
    # Calculating sqrt(a)
    bendingRootA = 0.246 - (3.08 * 10 ** -3) * Sut + (1.51 * 10 ** -5) * Sut ** 2 - (2.67 * 10 ** -8) * Sut ** 3

    # Calculating the notch sensitivity factor
    qBending = 1 / (1 + (bendingRootA / rootR))
    return 1 + qBending * (Kt - 1)


def kfs(rootR, Kts):
    """
    Calculates the torsional fatigue stress-concentration.
    """
    # Calculating sqrt(a)
    torsionalRootA = 0.190 - (2.51 * 10 ** -3) * Sut + (1.35 * 10 ** -5) * Sut ** 2 - (2.67 * 10 ** -8) * Sut ** 3
```

```
    # Calculating the notch sensitivity factor
    qTorsional = 1 / (1 + (torsionalRootA / rootR))
    return 1 + qTorsional * (Kts - 1)


def main():
    """
    Driving method for user inputs, and required calculations.

    1) Determines the problem type
    2) Calls dependent methods for calculations
    3) Displays the final safety factor, or diameter
    """
    while True:
        print("For the safety factor against fatigue using Goodman: 1")
        print("For the safety factor against first cycle yield using Von Mises stresses: 2")
        print("For the first cycle yield using conservative approximation: 3")
        print("For the first cycle yield using conservative approximation and first cycle yield using
the Goodman "
              "criteria: 4")
        print("To exit the program: 0")
        problemType = int(input("Problem type: "))

        if problemType == 1:
            moment, torque, diameter, Kt, Kts, rootR = get_user_input(problemType)
            result = goodman_criteria(moment, torque, diameter, Kt, Kts, rootR)
            print('\nThe factor of safety calculated from the Goodman criteria is: ' + str(round(result,
4)) + "\n")
        elif problemType == 2:
            moment, torque, diameter, Kt, Kts, rootR = get_user_input(problemType)
            result = vonmises_stress(moment, torque, diameter, Kt, Kts, rootR)
            print("The factor of safety calculated from the Von Mises stress is: " + str(round(result, 4))
+ "\n")
        elif problemType == 3:
            moment, torque, diameter, Kt, Kts, rootR = get_user_input(problemType)
            result = conservative_approximation(moment, torque, diameter, Kt, Kts, rootR)
            print("The factor of safety calculated from the Goodman criteria is: " + str(round(result,
4)) + "\n")
        elif problemType == 4:
            moment, torque, diameter, Kt, Kts, rootR = get_user_input(problemType)
            result = (infinite_life(moment, torque, diameter))
```

```
            print("The minimum diameter required is: " + str(round(result, 4)) + "\n")
        elif problemType == 0:
            break


if __name__ == "__main__":
    # Declaring material constants
    Sut = 68  # ksi
    Sy = 37.5
    sePrime = 0.5 * Sut

    # Calling driving method for the script
    main()
```