

AI Assisted Coding

Assignment 6.3

Name:M.Goutham

HT No: 2303A52010

Batch: 31

Examples:

1.

```
{  
  "personal": {  
    "full_name": "Naresh"  
  },  
  "academic": {  
    "branch": "CSE",  
    "sgpa": 8.0  
  }  
}
```

2.

```
{  
  "personal": {  
    "full_name": "Suresh"  
  },  
  "academic": {  
    "branch": "ECE",  
    "sgpa": 7.4  
  }  
}
```

1. Generate a Python function that prints all Perfect Numbers in a user given range, and regenerate an optimized version by checking divisors only up to square root of the number.

Examples:

1.

{

```
"personal": {  
    "full_name": "Naresh"  
},  
"academic": {  
    "branch": "CSE",  
    "sgpa": 8.0  
}
```

}

2.

{

```
"personal": {  
    "full_name": "Suresh"  
},  
"academic": {  
    "branch": "ECE",  
    "sgpa": 7.4  
}
```

}

```
import time
```

```
import re
```

```
from collections import Counter
```

1. Generate a Python function to display all Automorphic numbers between one and one thousand using a for loop. Regenerate the solution using a while loop and compare execution time.

```
# Automorphic Numbers (For Loop and While Loop)

def is_automorphic(n):
    """Check if a number is automorphic (n^2 ends with n)"""
    square = n * n
    return str(square).endswith(str(n))

def automorphic_for_loop():
    """Find automorphic numbers from 1 to 1000 using for loop"""
    result = []
    start_time = time.time()

    for num in range(1, 1001):
        if is_automorphic(num):
            result.append(num)

    end_time = time.time()
    execution_time = end_time - start_time

    print("Automorphic numbers (for loop):", result)
    print(f"Execution time (for loop): {execution_time:.6f} seconds")
    return result, execution_time

def automorphic_while_loop():
    """Find automorphic numbers from 1 to 1000 using while loop"""
    result = []
    start_time = time.time()

    num = 1
```

```

while num <= 1000:

    if is_automorphic(num):

        result.append(num)

    num += 1


end_time = time.time()

execution_time = end_time - start_time


print("Automorphic numbers (while loop):", result)

print(f"Execution time (while loop): {execution_time:.6f} seconds")

return result, execution_time

```

2. Generate Python code using nested if elif else conditions to classify online shopping feedback as Positive, Neutral, or Negative based on rating one to five. Also provide an alternative using dictionary or match case.

2. Shopping Feedback Classification

```

def classify_feedback_if_else(rating):

    """Classify feedback using nested if-elif-else"""

    if rating >= 1 and rating <= 5:

        if rating == 5 or rating == 4:

            return "Positive"

        elif rating == 3:

            return "Neutral"

        else: # rating == 1 or rating == 2

            return "Negative"

    else:

        return "Invalid rating"

```

```

def classify_feedback_dict(rating):

    """Classify feedback using dictionary"""

    feedback_map = {

```

```
    5: "Positive",
    4: "Positive",
    3: "Neutral",
    2: "Negative",
    1: "Negative"
}

return feedback_map.get(rating, "Invalid rating")
```

```
def classify_feedback_match(rating):
    """Classify feedback using match-case (Python 3.10+)"""

    match rating:
        case 5 | 4:
            return "Positive"
        case 3:
            return "Neutral"
        case 2 | 1:
            return "Negative"
        case _:
            return "Invalid rating"
```

3. Define a function named `statistical_operations` that computes minimum, maximum, mean, median, mode, variance, and standard deviation for a tuple of numbers.

3. Statistical Operations

```
def statistical_operations(numbers):
    """Compute statistical measures for a tuple of numbers"""

    import statistics

    if not numbers:
        return None

    results = {
```

```
"minimum": min(numbers),  
"maximum": max(numbers),  
"mean": statistics.mean(numbers),  
"median": statistics.median(numbers),  
"variance": statistics.variance(numbers) if len(numbers) > 1 else 0,  
"standard_deviation": statistics.stdev(numbers) if len(numbers) > 1 else 0  
}  
  
# Mode (handle cases where there might be no unique mode)
```

```
try:  
    results["mode"] = statistics.mode(numbers)  
except statistics.StatisticsError:  
    results["mode"] = "No unique mode"  
  
return results
```

4. Create a class named Teacher with attributes teacher_id, name, subject, and experience, and add a method to display teacher details.

4. Teacher Class

```
class Teacher:  
  
    def __init__(self, teacher_id, name, subject, experience):  
        self.teacher_id = teacher_id  
        self.name = name  
        self.subject = subject  
        self.experience = experience  
  
    def display_details(self):  
        """Display teacher details"""  
        print(f"Teacher ID: {self.teacher_id}")
```

```
print(f"Name: {self.name}")  
print(f"Subject: {self.subject}")  
print(f"Experience: {self.experience} years")
```

5. Generate a Python function to validate an Indian mobile number that starts with six, seven, eight, or nine and contains exactly ten digits.

Indian Mobile Number Validator

```
def validate_indian_mobile(number):  
    """Validate Indian mobile number (10 digits, starts with 6, 7, 8, or 9)"""  
  
    # Remove any spaces or hyphens  
  
    number = str(number).replace(" ", "").replace("-", "")  
  
  
    # Check using regex  
  
    pattern = r'^[6-9]\d{9}$'  
  
  
    if re.match(pattern, number):  
        return True  
  
    return False
```

6. Write a function to find all Armstrong numbers in a user specified range using a for loop and digit power logic. Regenerate an optimized version using list comprehension.

Armstrong Numbers

```
def is_armstrong(n):  
    """Check if a number is an Armstrong number"""  
  
    digits = str(n)  
  
    power = len(digits)  
  
    return n == sum(int(digit) ** power for digit in digits)
```

```
def armstrong_numbers_for_loop(start, end):
    """Find Armstrong numbers using for loop"""
    result = []
    for num in range(start, end + 1):
        digits = str(num)
        power = len(digits)
        total = 0
        for digit in digits:
            total += int(digit) ** power
        if total == num:
            result.append(num)
    return result
```

```
def armstrong_numbers_list_comp(start, end):
    """Find Armstrong numbers using list comprehension (optimized)"""
    return [n for n in range(start, end + 1) if n == sum(int(d) ** len(str(n)) for d in str(n))]
```

7. Generate a function that displays all Happy Numbers within a given range using loop logic, and regenerate an optimized version using a set to detect cycles.

```
# Happy Numbers
```

```
def is_happy_basic(n):
    """Check if number is happy using basic loop"""
    seen = []
    while n != 1 and n not in seen:
        seen.append(n)
        n = sum(int(digit) ** 2 for digit in str(n))
    return n == 1
```

```
def is_happy_optimized(n):
    """Check if number is happy using set (optimized)"""
```

```

seen = set()

while n != 1 and n not in seen:
    seen.add(n)
    n = sum(int(digit) ** 2 for digit in str(n))

return n == 1


def happy_numbers_basic(start, end):
    """Find happy numbers using basic loop logic"""

    result = []

    for num in range(start, end + 1):
        if is_happy_basic(num):
            result.append(num)

    return result


def happy_numbers_optimized(start, end):
    """Find happy numbers using set for cycle detection"""

    return [n for n in range(start, end + 1) if is_happy_optimized(n)]

```

8. Generate a function to display all Strong Numbers within a given range using loops, and regenerate an optimized version using precomputed factorials.

Strong Numbers

```

def factorial(n):
    """Calculate factorial"""

    if n == 0 or n == 1:
        return 1

    result = 1

    for i in range(2, n + 1):
        result *= i

    return result

```

```
def is_strong_basic(n):
    """Check if number is strong using loops"""
    total = 0
    for digit in str(n):
        total += factorial(int(digit))
    return total == n

# Precompute factorials for optimization
FACTORIAL_CACHE = {i: factorial(i) for i in range(10)}

def is_strong_optimized(n):
    """Check if number is strong using precomputed factorials"""
    return sum(FACTORIAL_CACHE[int(digit)] for digit in str(n)) == n

def strong_numbers_basic(start, end):
    """Find strong numbers using loops"""
    result = []
    for num in range(start, end + 1):
        if is_strong_basic(num):
            result.append(num)
    return result

def strong_numbers_optimized(start, end):
    """Find strong numbers using precomputed factorials"""
    return [n for n in range(start, end + 1) if is_strong_optimized(n)]
```

Extract Student Information

```

def extract_student_info(student_dict):
    """Extract full name, branch, and SGPA from nested dictionary"""
    try:
        full_name = student_dict["personal"]["full_name"]
        branch = student_dict["academic"]["branch"]
        sgpa = student_dict["academic"]["sgpa"]

    return {
        "full_name": full_name,
        "branch": branch,
        "sgpa": sgpa
    }

except KeyError as e:
    return f"Missing key: {e}"

```

10. Generate a Python function that prints all Perfect Numbers in a user given range, and regenerate an optimized version by checking divisors only up to square root of the number.

Perfect Numbers

```

def is_perfect_basic(n):
    """Check if number is perfect (sum of divisors equals n)"""

    if n < 2:
        return False

    divisor_sum = 1 # 1 is always a divisor

    for i in range(2, n):
        if n % i == 0:
            divisor_sum += i

    return divisor_sum == n

```

```
def is_perfect_optimized(n):
    """Check if number is perfect (optimized - check up to sqrt)"""
    if n < 2:
        return False
    divisor_sum = 1
    i = 2
    while i * i <= n:
        if n % i == 0:
            divisor_sum += i
            if i * i != n: # Add the paired divisor
                divisor_sum += n // i
        i += 1
    return divisor_sum == n
```

```
def perfect_numbers_basic(start, end):
    """Find perfect numbers using basic method"""
    result = []
    for num in range(start, end + 1):
        if is_perfect_basic(num):
            result.append(num)
    return result
```

```
def perfect_numbers_optimized(start, end):
    """Find perfect numbers using optimized method"""
    return [n for n in range(start, end + 1) if is_perfect_optimized(n)]
```

```
# =====
# TEST ALL FUNCTIONS
# =====
```

```
if __name__ == "__main__":
    print("=" * 60)
    print("1. AUTOMORPHIC NUMBERS (1 to 1000)")
    print("=" * 60)
    automorphic_for_loop()
    print()
    automorphic_while_loop()

    print("\n" + "=" * 60)
    print("2. SHOPPING FEEDBACK CLASSIFICATION")
    print("=" * 60)
    for rating in range(1, 6):
        print(f"Rating {rating}: {classify_feedback_if_else(rating)} (if-else), "
              f"{classify_feedback_dict(rating)} (dict), "
              f"{classify_feedback_match(rating)} (match)")

    print("\n" + "=" * 60)
    print("3. STATISTICAL OPERATIONS")
    print("=" * 60)
    test_numbers = (10, 20, 30, 40, 50, 30)
    stats = statistical_operations(test_numbers)
    for key, value in stats.items():
        print(f"{key.capitalize()}: {value}")

    print("\n" + "=" * 60)
    print("4. TEACHER CLASS")
    print("=" * 60)
    teacher1 = Teacher("T001", "Dr. Smith", "Mathematics", 15)
    teacher1.display_details()
```

```
print("\n" + "=" * 60)
print("5. INDIAN MOBILE NUMBER VALIDATION")
print("=" * 60)

test_numbers_mobile = ["9876543210", "8123456789", "5123456789", "98765432"]

for num in test_numbers_mobile:
    print(f"{num}: {'Valid' if validate_instant(num) else 'Invalid'}")

print("\n" + "=" * 60)
print("6. ARMSTRONG NUMBERS (1 to 500)")

print("=" * 60)
print("For loop:", armstrong_numbers_for_loop(1, 500))
print("List comprehension:", armstrong_numbers_list_comp(1, 500))

print("\n" + "=" * 60)
print("7. HAPPY NUMBERS (1 to 100)")

print("=" * 60)
print("Basic:", happy_numbers_basic(1, 100))
print("Optimized:", happy_numbers_optimized(1, 100))

print("\n" + "=" * 60)
print("8. STRONG NUMBERS (1 to 200)")

print("=" * 60)
print("Basic:", strong_numbers_basic(1, 200))
print("Optimized:", strong_numbers_optimized(1, 200))

print("\n" + "=" * 60)
print("9. EXTRACT STUDENT INFORMATION")
print("=" * 60)

student1 = {
    "personal": {"full_name": "Naresh"},
```

```
"academic": {"branch": "CSE", "sgpa": 8.0}  
}  
  
student2 = {  
    "personal": {"full_name": "Suresh"},  
    "academic": {"branch": "ECE", "sgpa": 7.4}  
}  
  
print("Student 1:", extract_student_info(student1))  
print("Student 2:", extract_student_info(student2))  
  
  
print("\n" + "=" * 60)  
print("10. PERFECT NUMBERS (1 to 10000)")  
print("=" * 60)  
print("Basic:", perfect_numbers_basic(1, 10000))  
print("Optimized:", perfect_numbers_optimized(1, 10000))
```

Output:

```
=====
```

1. AUTOMORPHIC NUMBERS (1 to 1000)

```
=====
```

Automorphic numbers (for loop): [1, 5, 6, 25, 76, 376, 625]

Execution time (for loop): 0.000361 seconds

Automorphic numbers (while loop): [1, 5, 6, 25, 76, 376, 625]

Execution time (while loop): 0.000359 seconds

```
=====
```

2. SHOPPING FEEDBACK CLASSIFICATION

```
=====
```

Rating 1: Negative (if-else), Negative (dict), Negative (match)

Rating 2: Negative (if-else), Negative (dict), Negative (match)

Rating 3: Neutral (if-else), Neutral (dict), Neutral (match)

Rating 4: Positive (if-else), Positive (dict), Positive (match)

Rating 5: Positive (if-else), Positive (dict), Positive (match)

3. STATISTICAL OPERATIONS

Minimum: 10

Maximum: 50

Mean: 30

Median: 30.0

Variance: 200

Standard_deviation: 14.142135623730951

Mode: 30

4. TEACHER CLASS

Teacher ID: T001

Name: Dr. Smith

Subject: Mathematics

Experience: 15 years

5. INDIAN MOBILE NUMBER VALIDATION

9876543210: Valid

8123456789: Valid

5123456789: Invalid

98765432: Invalid

=====

6. ARMSTRONG NUMBERS (1 to 500)

=====

For loop: [1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407]

List comprehension: [1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407]

=====

7. HAPPY NUMBERS (1 to 100)

=====

Basic: [1, 7, 10, 13, 19, 23, 28, 31, 32, 44, 49, 68, 70, 79, 82, 86, 91, 94, 97, 100]

Optimized: [1, 7, 10, 13, 19, 23, 28, 31, 32, 44, 49, 68, 70, 79, 82, 86, 91, 94, 97, 100]

=====

8. STRONG NUMBERS (1 to 200)

=====

Basic: [1, 2, 145]

Optimized: [1, 2, 145]

=====

9. EXTRACT STUDENT INFORMATION

=====

Student 1: {'full_name': 'Naresh', 'branch': 'CSE', 'sgpa': 8.0}

Student 2: {'full_name': 'Suresh', 'branch': 'ECE', 'sgpa': 7.4}

=====

10. PERFECT NUMBERS (1 to 10000)

=====

Basic: [6, 28, 496, 8128]

Explanations:

1. The program finds Automorphic numbers by checking whether the square of a number ends with the number itself using loop logic.
2. The feedback classification uses conditional statements to correctly label ratings as Negative, Neutral, or Positive based on given values.
3. The function uses Python built-in statistics methods to compute minimum, maximum, mean, median, mode, variance, and standard deviation accurately.
4. The Teacher class demonstrates object oriented programming by initializing attributes through a constructor and displaying details using a class method.
5. The function validates an Indian mobile number by checking that it has exactly ten digits and starts with six, seven, eight, or nine.
6. The program identifies Armstrong numbers by comparing each number with the sum of its digits raised to the power of total digits.
7. Happy numbers are detected by repeatedly summing the squares of digits and using a set to prevent infinite loops.
8. The function checks whether a number equals the sum of factorials of its digits to identify Strong numbers.
9. The function navigates a nested dictionary structure to correctly extract student full name, branch, and SGPA.
10. Perfect numbers are identified by summing proper divisors efficiently by checking only up to the square root of the number.