# Lab Manual

## Practical and Skills Development

# CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

| | |
|---|---|
| **Registration No** | : 25BCE10435 |
| **Name of Student** | : ANIKET PORWAL |
| **Course Name** | : Introduction to Problem Solving and Programming |
| **Course Code** | : CSE1021 |
| **School Name** | : SCOPE |
| **Slot** | : B11+B12+B13 |
| **Class ID** | : BL2025260100796 |
| **Semester** | : FALL 2025/26 |

Course Faculty Name : Dr. Hemraj S. Lamkuche

Signature:

VIT BHOPAL

# Practical Index

| S. No. | Title of Practical | Date of Submission | Signature of Faculty |
|---|---|---|---|
| 1 | Write a function factorial(n) that calculates the factorial of a non-negative integer n (n!). | 05-10-25 | |
| 2 | Write a function is_palindrome(n) that checks if a number reads the same forwards and backwards. | 05-10-25 | |
| 3 | Write a function mean_of_digits(n) that returns the average of all digits in a number. | 05-10-25 | |
| 4 | Write a function digital_root(n) that repeatedly sums the digits of a number until a single digit is obtained. | 05-10-25 | |
| 5 | Write a function is_abundant(n) that returns True if the sum of proper divisors of n is greater than n. | 05-10-25 | |
| 6 | Write a function is_deficient(n) that returns True if the sum of proper divisors of n is less than n. | 02-11-25 | |
| 7 | Write a function for Harshad number is_harshad(n) that checks if a number is divisible by the sum of its digits. | 02-11-25 | |
| 8 | Write a function is_automorphic(n) that checks if a number's square ends with the number itself. | 02-11-25 | |
| 9 | Write a function is_pronic(n) that checks if a number is the product of two consecutive integers. | 02-11-25 | |
| 10 | Write a function prime_factors(n) that returns the list of prime factors of a number. | 02-11-25 | |
| 11 | Write a function count_distinct_prime_factors(n) that returns how many unique prime factors a number has. | 09-11-25 | |
| 12 | Write a function is_prime_power(n) that checks if a number can be expressed as pk where p is prime and k ≥ 1. | 09-11-25 | |
| 13 | Write a function is_mersenne_prime(p) that checks if 2p - 1 is a prime number (given that p is prime). | 09-11-25 | |
| 14 | Write a function twin_primes(limit) that generates all twin prime pairs up to a given limit. | 09-11-25 | |

|  |  |  |  |
|---|---|---|---|
| **15** | **Write a function Number of Divisors (d(n)) count_divisors(n) that returns how many positive divisors a number has.** | **09-11-25** |  |

**Practical No: 11**

**Date: 09-11-25**

**TITLE**: Program to check how many unique prime factors a number has.

**AIM/OBJECTIVE(s)**: **Write a function count_distinct_prime_factors(n) that returns how many unique prime factors a number has.**

**METHODOLOGY & TOOL USED**:

Methodology:

1. **Start from the smallest prime (2):** Begin checking divisibility from 2 up to the square root of n.

2. **Trial Division:** For each number i, check if n is divisible by i. If it is, then i is a prime factor. Increment the count of distinct prime factors.

3. **Remove Repeated Factors:** Divide n repeatedly by i until it's no longer divisible — this ensures each prime factor is only counted once.

4. **Check Remaining Value:** After the loop, if n > 1, then n itself is a prime factor (because what remains is prime).

5. **Return the Count:** Finally, return the total number of distinct prime factors found.

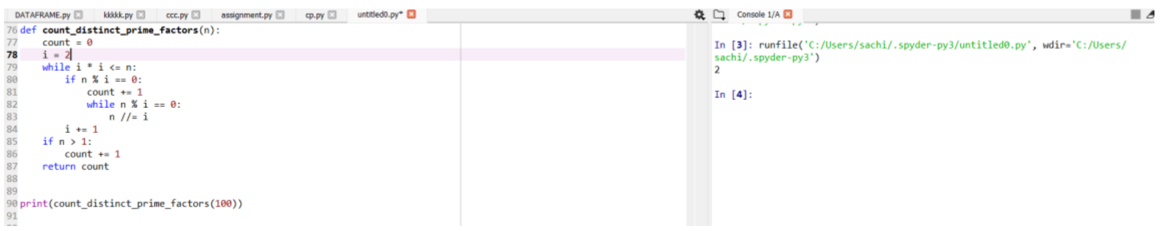
Tools Used:

- Looping (while loop) – to iterate through possible factors.
- Conditional statements (if) – to check divisibility.
- Mathematical logic – based on properties of prime numbers and factorization.

## BRIEF DESCRIPTION:

The function count_distinct_prime_factors(n) is designed to find and count the number of unique prime factors of a given integer n. It works by repeatedly checking which numbers divide n starting from 2 (the smallest prime number). Each time it finds a number that divides n, it counts it as one distinct prime factor and then divides n completely by that factor to remove all of its multiples. After checking up to the square root of n, if the remaining value of n is greater than 1, that means it is itself a prime number, and it is counted as one more distinct prime factor. Finally, the function returns the total count of these unique prime factors.

## RESULTS ACHIEVED:



```python
def count_distinct_prime_factors(n):
    count = 0
    i = 2
    while i * i <= n:
        if n % i == 0:
            count += 1
            while n % i == 0:
                n //= i
        i += 1
    if n > 1:
        count += 1
    return count


print(count_distinct_prime_factors(100))
```

```
In [3]: runfile('C:/Users/sachi/.spyder-py3/untitled0.py', wdir='C:/Users/
sachi/.spyder-py3')
2

In [4]:
```

## DIFFICULTY FACED BY STUDENT:

1. Understanding Prime Factorization Logic: Many students struggle to grasp why the loop only goes up to the square root of n, and not all the way to n.

2. Handling Repeated Prime Factors: Students often forget to divide n repeatedly by the same factor (inside the inner while loop), causing repeated counting of the same prime factor.

3. Identifying Remaining Prime Factor: Some students miss the final check (if n > 1), which accounts for the last prime factor left after all divisions.

4. Confusing Prime and Composite Numbers: Beginners sometimes mix up prime factors with all factors, leading to incorrect outputs.

5. Integer Division Errors: Forgetting to use integer division (//) instead of normal division (/) may cause type errors or incorrect logic in Python.

**SKILLS ACHIEVED**:

1. Understanding of Prime Factorization: Students learn how to break down a number into its prime components — an essential concept in number theory.

2. Logical Thinking and Problem Solving: The step-by-step process of identifying and counting distinct prime factors enhances logical reasoning.

3. Efficient Loop and Condition Use: Practice in using loops (while) and conditional statements (if) effectively to implement mathematical logic.

4. Optimization Awareness: Learning why the loop runs only up to the square root of n builds awareness of algorithm efficiency.

5. Programming Fundamentals: Improves understanding of integer operations, modular arithmetic (%), and iterative problem-solving in Python.

**Practical No: 12**

**Date: 09-11-25**

**TITLE: Program to check whether a number is prime power number.**

**AIM/OBJECTIVE(s)**: **Write a function is_prime_power(n) that checks if a number can be expressed as pk where p is prime and k ≥ 1.**

**METHODOLOGY & TOOL USED**:

**Methodology: -**

1. Concept Used: The function is based on the mathematical concept of prime powers, where a number can be written as —

here, p is a prime number and k ≥ 1 is an integer exponent.

2. Step-by-Step Approach:

Step 1: Use a helper function is_prime (x) to check whether a number is prime. It checks divisibility from 2 to √x. If no divisor is found, x is prime.

Step 2: Loop through all numbers p from 2 to √n. Check if p is prime.
Step 3: For each prime p, raise it to successive powers p^k (where k ≥ 1) until the power exceeds or equals n.
Step 4: If for any p, p^k == n, then n is a prime power, return True. Otherwise, after all checks, return False.

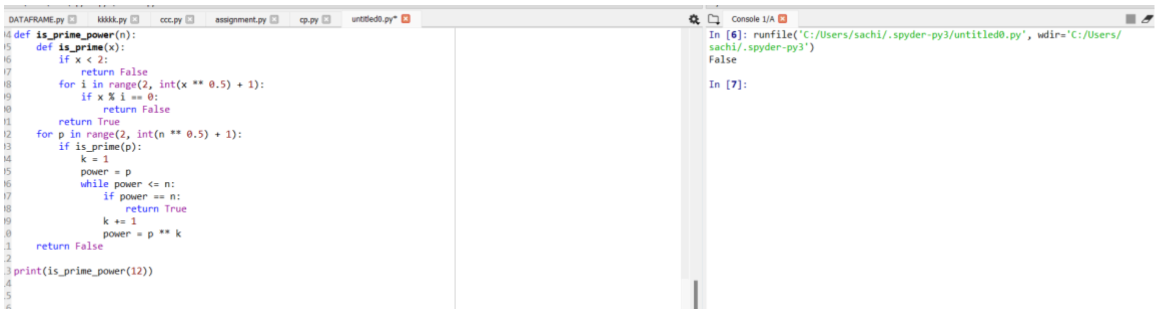
**Tool: -**

- Mathematical logic (prime and exponentiation).
- Python programming constructs: loops, conditional statements, and function definitions.
- Optimization using square root bound (√n) to reduce iterations.

**BRIEF DESCRIPTION**:

The function is_prime_power(n) checks whether a given number n can be written as a power of a prime number. It first identifies all possible prime bases (p) and then raises each to successive powers until the value equals or exceeds n. If at any point, the function returns True, meaning that n is a prime power. Otherwise, it returns False. This function helps in understanding the mathematical relationship between numbers and their prime components using logical iteration and power computation in Python.

**RESULTS ACHIEVED**:



```python
def is_prime_power(n):
    def is_prime(x):
        if x < 2:
            return False
        for i in range(2, int(x ** 0.5) + 1):
            if x % i == 0:
                return False
        return True
    for p in range(2, int(n ** 0.5) + 1):
        if is_prime(p):
            k = 1
            power = p
            while power <= n:
                if power == n:
                    return True
                k += 1
                power = p ** k
    return False

print(is_prime_power(12))
```

```
In [6]: runfile('C:/Users/sachi/.spyder-py3/untitled0.py', wdir='C:/Users/
sachi/.spyder-py3')
False

In [7]:
```

**DIFFICULTY FACED BY STUDENT**:

1. Understanding the Concept of Prime Powers: Many students initially find it hard to grasp that a number can be expressed as  only if p is prime and k ≥ 1.

2. Implementing Prime Checking Correctly: Writing an efficient and accurate is_prime() function can be confusing — especially handling edge cases like 0, 1, or negative numbers.

3. Loop Logic and Power Calculation: Students may struggle with setting correct loop limits (using √n) and updating the power (p ** k) properly without causing infinite loops.

4. Distinguishing Between Prime and Composite Powers: Beginners often mistake composite bases (like $4 = 2^2$) for primes, which leads to incorrect results.

5. Optimization and Efficiency: Understanding why checking primes only up to √n is enough can be a challenge for new learners.

**SKILLS ACHIEVED**:

1. Students learn how to identify prime numbers and express other numbers as powers of primes.
2. The task develops analytical skills by requiring step-by-step reasoning to test different prime bases and exponents.
3. Students practice using loops, conditional statements, and helper functions effectively.
4. This exercise strengthens the connection between mathematical theory (prime factorization) and practical coding implementation.
5. Learners improve their debugging skills by handling edge cases (like $n = 1$ or small primes) and optimizing the code to avoid unnecessary calculations.

**Practical No: 13**

**Date: 09-11-25**

**TITLE**: Program to check whether a number is a Mersenne prime number.

**AIM/OBJECTIVE(s)**: **Write a function is_mersenne_prime(p) that checks if 2p – 1 is a prime number (given that p is prime).**

**METHODOLOGY & TOOL USED**:

Methodology: -

1. Concept Used: A Mersenne prime is a prime number of the form, where p itself is a prime number.

2. Step-by-Step Process:

Step 1: Create a helper function is_prime(n) to test whether a number is prime.

Step 2: Verify that p (the exponent) is prime — because Mersenne primes only exist for prime values of p.

Step 3: Compute the Mersenne number as.

Step 4: Check if the result is also a prime number.

Step 5: Return True if both are prime, otherwise False.

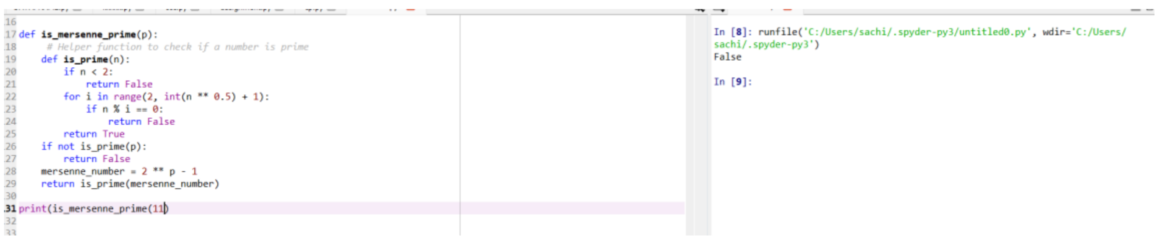
Tools: -

- Mathematical reasoning using the definition of Mersenne primes.
- Python functions and loops for checking primality.
- Optimization with the square root method for efficient prime testing.

**BRIEF DESCRIPTION**:

The function is_mersenne_prime(p) checks if a number of the form is a Mersenne prime. It first ensures that p is a prime number and then verifies whether is also prime. If both conditions are satisfied, it returns True; otherwise, False. This function combines mathematical logic with programming to identify a special type of prime number.

**RESULTS ACHIEVED**:



```python
def is_mersenne_prime(p):
    # Helper function to check if a number is prime
    def is_prime(n):
        if n < 2:
            return False
        for i in range(2, int(n ** 0.5) + 1):
            if n % i == 0:
                return False
        return True
    if not is_prime(p):
        return False
    mersenne_number = 2 ** p - 1
    return is_prime(mersenne_number)

print(is_mersenne_prime(11)
```

```
In [8]: runfile('C:/Users/sachi/.spyder-py3/untitled0.py', wdir='C:/Users/
sachi/.spyder-py3')
False

In [9]:
```

**DIFFICULTY FACED BY STUDENT**:

1. Confusion about checking p being prime before calculating.

2. Handling large exponent values that make computations slower.

3. Understanding the difference between Mersenne numbers and Mersenne primes.

4. Implementing an efficient and correct prime checking function.

5. Managing data types and precision for large powers in Python.

**SKILLS ACHIEVED**:

1. Understanding Mersenne primes and their mathematical properties.

2. Improving skills in nested function use and prime checking algorithms.

3. Applying mathematical theory to programming problems.

4. Learning modular coding by separating logic into helper functions.

5. Strengthening problem-solving and logical reasoning abilities.

**Practical No: 14**

**Date: 09-11-25**

**TITLE**: To check whether a number is a twin prime number.

**AIM/OBJECTIVE(s): Write a function twin_primes(limit) that generates all twin prime pairs up to a given limit.**

**METHODOLOGY & TOOL USED**:

Methodology: -

1. Prime Checking Approach: A helper function is_prime(n) is used to verify if a number is prime. It checks divisibility from 2 up to the square root of n. If n is divisible by any number in this range, it is not prime.

2. Twin Prime Generation: Loop through numbers starting from 2 up to limit - 1. For each number i, check if both i and i + 2 are prime. If true, store the pair (i, i + 2) in a list.

3. Result Compilation: All such pairs are collected in a list twins. The list is returned at the end containing all twin prime pairs up to the given limit.

4. Iterative & Conditional Logic: The process combines iteration (for checking all numbers) and conditional testing (for primality and twin property).

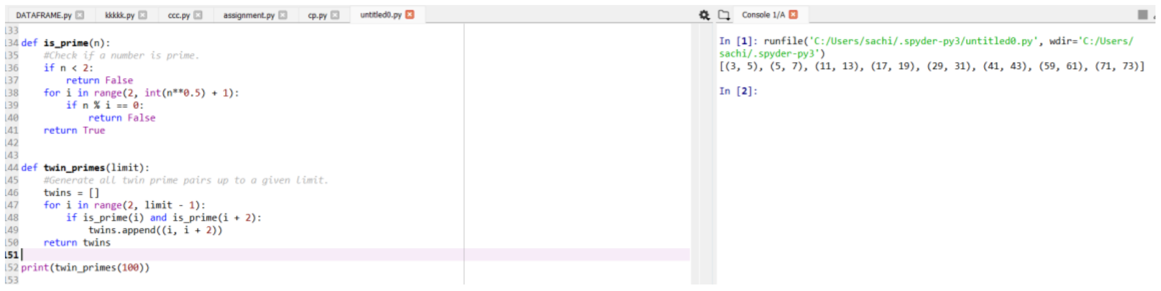
Tools: -

- Looping (for loop): To iterate through numbers from 2 up to the given limit.
- Conditional Statements (if): To check whether both numbers in the pair are prime.
- Mathematical Operations: Square root calculation using n**0.5 for efficient prime checking.
- List Data Structure: To store and return the list of twin prime pairs.
- Function Definition: is_prime(n) for prime checking. twin_primes(limit) for generating twin primes.

**BRIEF DESCRIPTION**:

This program is designed to generate all twin prime pairs up to a given limit. A twin prime is a pair of prime numbers that differ by exactly 2, such as (3, 5) or (11, 13). The program works by first defining a helper function is_prime(n) to check if a number is prime. Then, in the main function twin_primes(limit), it loops through all numbers up to the given limit and checks whether both the current number i and i + 2 are prime. If they are, the pair is added to a list of twin primes. Finally, it returns the list of all twin prime pairs within the range specified by the user.

**RESULTS ACHIEVED**:



DATAFRAME.py kkkkk.py ccc.py assignment.py cp.py untitled0.py

```python
def is_prime(n):
    #Check if a number is prime.
    if n < 2:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True


def twin_primes(limit):
    #Generate all twin prime pairs up to a given limit.
    twins = []
    for i in range(2, limit - 1):
        if is_prime(i) and is_prime(i + 2):
            twins.append((i, i + 2))
    return twins

print(twin_primes(100))
```

Console 1/A

```
In [1]: runfile('C:/Users/sachi/.spyder-py3/untitled0.py', wdir='C:/Users/
sachi/.spyder-py3')
[(3, 5), (5, 7), (11, 13), (17, 19), (29, 31), (41, 43), (59, 61), (71, 73)]

In [2]:
```

**DIFFICULTY FACED BY STUDENT**:

1. Understanding the Concept of Twin Primes: Many students initially get confused between prime numbers and twin prime pairs. They may not realize that twin primes must differ by exactly

2. Writing the Prime Check Function: Implementing an efficient is_prime () function can be tricky. Students sometimes forget to check divisibility only up to the square root of n, leading to slower or incorrect results.

3. Loop Range Confusion: Some students struggle with setting the correct loop range (limit - 1 instead of limit) and may accidentally go out of bounds when checking i + 2.

4. Logical Errors in Pair Formation: It's common to mistakenly print or store only one number instead of a pair (i, i + 2).

5. Handling Small Limits: When the limit is too small (like below 5), students might not handle the case properly and expect output when no twin primes exist.

**SKILLS ACHIEVED**:

- Students strengthen their understanding of prime numbers and their properties.
- Developing the twin_primes () function helps improve logical reasoning and structured problem-solving skills.
- Students learn how to define and use multiple functions (is_prime () and twin_primes ()) to break down a problem into smaller parts.
- Practice in using loops and conditional statements effectively for numerical computations.

**Practical No: 15**

**Date: 02-11-25**

**TITLE**: Program to find number of positive divisors a number have.

**AIM/OBJECTIVE(s):** Write a function Number of Divisors (d(n)) count_divisors(n) that returns how many positive divisors a number has.

**METHODOLOGY & TOOL USED**:

Methodology: -

1. Mathematical Logic: Every divisor i less than or equal to √n has a corresponding divisor n // i. So, for each divisor found, we count both i and n//i.

2. Optimization: Instead of looping up to n, we only loop up to √n to make the function more efficient.

3. Condition for Perfect Squares: If n is a perfect square (e.g., 16 → 4×4), we count the divisor only once.

4. Iteration and Counting: We increment the count for each valid divisor pair.

Tools: -

Programming Language: Python

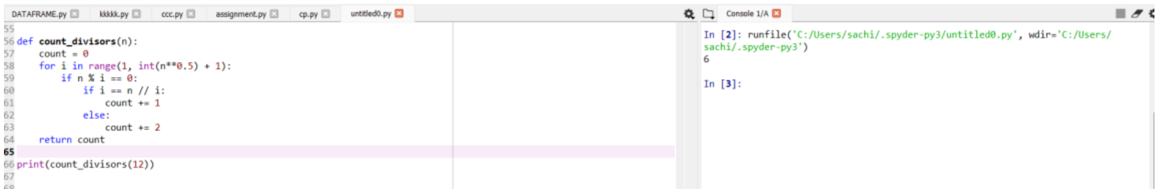
Concepts Used:

- for loop for iteration
- % (modulus) operator for divisibility check
- **0.5 for square root calculation
- Conditional statements to handle perfect squares

**BRIEF DESCRIPTION**:

This program defines a function count_divisors(n) that counts how many positive integers divide n completely. It efficiently calculates the total number of divisors by checking up to the square root of n, counting each divisor pair, and handling perfect squares carefully.

**RESULTS ACHIEVED**:



```python
def count_divisors(n):
    count = 0
    for i in range(1, int(n**0.5) + 1):
        if n % i == 0:
            if i == n // i:
                count += 1
            else:
                count += 2
    return count

print(count_divisors(12))
```

```
In [2]: runfile('C:/Users/sachi/.spyder-py3/untitled0.py', wdir='C:/Users/
sachi/.spyder-py3')
6

In [3]:
```

**DIFFICULTY FACED BY STUDENT**:

1. Forgetting to include both i and n//i as divisors.

2. Not handling perfect squares correctly (double-counting one divisor).

3. Looping all the way up to n instead of √n, making the code inefficient.

4. Confusion between factors and divisors (they are the same in this context).

**SKILLS ACHIEVED**:

- Understanding of divisors and factorization.
- Use of mathematical optimization (√n approach).
- Strengthened problem-solving and loop control skills.
- Ability to design efficient and accurate mathematical functions in Python.