

NAME : KODITHYALA AKSHITH

ROLL NO: 2303A52436

BATCH: 44

Task-1

The screenshot shows a code editor interface with a Python file named 'lab1-4.py' open. The code is a simple prime-checking script using user input and no functions. It includes error handling for non-integer input. The code editor has a sidebar for 'BLACKBOX Agent' and a terminal window at the bottom showing the script's execution and output.

```
lab1-4.py > ...
1 # Prime check using user input, no functions
2
3 n_str = input("Enter a positive integer: ")
4 try:
5     n = int(n_str)
6 except:
7     print("Invalid input. Please enter an integer.")
8     exit()
9
10 if n <= 1:
11     print(f"{n} is not a prime number.")
12 else:
13     is_prime = True
14     i = 2
15     while i * i <= n:
16         if n % i == 0:
17             is_prime = False
18             break
19         i += 1
20
21     if is_prime:
22         print(f"{n} is a prime number.")
23     else:
24         print(f"{n} is not a prime number.")

# generate a list co... 23 hr ago
```

generate a prime number logic without using function and using user-input

Used 1 reference
GitHub Copilot

Here's a simple prime-checking script using user input and no function definitions. Place it in /Users/akshithkodithyala/Desktop/ai asst coding/lab1-4.py.

...existing code...
Prime check using user input, no fu
n_str = input("Enter a positive integ
try:
 n = int(n_str)
except:
 print("Invalid input. Please ente
 exit()

if n <= 1:
 print(f"{n} is not a prime number
else:
 is_prime = True
 i = 2
 while i * i <= n:
 if n % i == 0:
 is_prime = False
 break
 i += 1

 if is_prime:
 print(f"{n} is a prime number
 else:
 print(f"{n} is not a prime nu
...existing code...

Add Context... lab1-4.py X
Add context (#), extensions (@), commands
Ask v GPT-5 mini v Go Live BLACKBOXAI: Open Chat

PROMPTS:

Generate a prime number logic without using function and using user-input

Input: 5

Output: 5 is a prime number

Input: 25

Output: 25 is mot a prime number

Task-2

The screenshot shows a code editor interface with several panels:

- Left Panel:** Displays the original Python code for prime number checking. It includes imports, variable declarations, and a loop that checks for divisibility from 2 to n-1.
- Right Panel:** Shows the optimized code where the loop is modified to skip even numbers (i += 2) and includes a condition to break if a divisor is found.
- Bottom Panel:** A terminal window shows the execution of both the original and optimized code on the same input (n=21), demonstrating that they produce identical results.
- Bottom Right:** A sidebar provides context about the code, such as "Used 1 reference" and "Optimized prime check using 6k±1 rule (no functions, user input)".

PROMPT:

Optimize prime number checking logic

Original code runs the loop to n times on checking the condition one by one

Since Optimized code runs skipping the odd divisors reducing the number of loops

Task-3

The screenshot shows a code editor interface with two tabs: 'lab1-4.py' and 'lab1-4.py'. The left pane displays the code for 'lab1-4.py', which contains a function to check if a number is prime and a main function that prompts the user for input and prints whether the number is prime or not. The right pane shows the same code with some parts highlighted in blue, indicating they are part of a reference solution. The terminal at the bottom shows the execution of the script and its output.

```
# Prime number check using functions
def is_prime(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True

def main():
    n_str = input("Enter a positive integer: ").strip()
    try:
        n = int(n_str)
    except ValueError:
        print("Invalid input. Please enter an integer.")
        return
    if is_prime(n):
        print(f"{n} is a prime number.")
    else:
        print(f"{n} is not a prime number.")

if __name__ == "__main__":
    main()
```

```
# ...existing code...
# Prime number check using functions
def is_prime(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True

def main():
    n_str = input("Enter a positive integer: ")
    try:
        n = int(n_str)
    except ValueError:
        print("Invalid input. Please enter an integer.")
        return
    if is_prime(n):
        print(f"{n} is a prime number.")
    else:
        print(f"{n} is not a prime number.")

if __name__ == "__main__":
    main()
# ...existing code...
```

PROMPT:

Prime Number Check Using Functions

Code:

```
def is_prime(num: int) -> bool:

    if num <= 1:
        return False

    if num == 2:
        return True

    if num % 2 == 0:
        return False

    for i in range(3, int(num ** 0.5) + 1, 2):
        if num % i == 0:
            return False

    return True
```

Test Cases:

Input: 5

Output: 5 is a prime number

Input: 2

Output: 2 is a prime number

Input: 15

Output: 15 is not a prime number

Task-4

Comparison Table:

The screenshot shows a code editor interface with a Python file named 'lab1-4.py'. The code implements a prime check function using both functional and imperative approaches. A comparison table is overlaid on the right side of the editor, titled 'Comparative Analysis - With vs Without Functions'.

Aspect	With functions	Without functions	Recommendation
Readability	High — logic separated, name documents intent (<code>is_prime</code>)	Lower — logic embedded in flow	Use functions for clarity
Reusability	High — call <code>is_prime()</code> from other code/test	Low — must copy/paste logic to reuse	Prefer functions when reuse likely
Testability	Easy — unit tests target <code>is_prime</code>	Harder — must run program and parse output	Functions are better for automated tests
Maintainability	Easier — single place to update algorithm	Harder — changes duplicated if used elsewhere	Functions reduce duplication
Performance	Same algorithmic complexity ($O(\sqrt{n})$), small call overhead per call	Same complexity; tiny win avoiding call overhead	Call overhead usually negligible; choose clarity
Memory & stack	Slight stack use for call frame	Slightly less stack	Only matters in extreme microbenchmarks
Scalability	Better — can be used in loops, libraries, multiprocessing	Worse — inline code less modular	Functions scale better for larger projects
Debugging	Easier to isolate and log inside function	Can be messier in long scripts	Prefer functions for debugging
File size / LOC	Slightly more lines but organized	Fewer lines, flatter	Prefer structured code unless one-off quick script
Best use-case	Libraries, programs with tests, reuse, clarity	One-off throwaway scripts or educational demos	Use functions in production/assignments

Below the table, a 'Short analytical summary' section provides a bulleted list of observations:

- Both approaches implement the same $O(\sqrt{n})$ algorithm and have identical time complexity $O(\sqrt{n})$.
- Using a function (`is_prime`) improves readability, reusability, testability, and maintainability with negligible runtime cost for typical use.
- An inline (no-function) version can be marginally faster due to avoided call overhead, but that difference is trivial except in microbenchmarks called.

At the bottom of the editor, there are context menus for 'Add Context...', 'Ask', and 'GPT-5 mini'.

TASK-5

```
# Prime number check using functions
def is_prime():
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True

def is_prime_miller_rabin(n):
    if n < 2:
        return False
    small_primes = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
    for p in small_primes:
        if n % p == 0:
            return n == p
    d = n - 1
    s = 0
    while d % 2 == 0:
        d /= 2
        s += 1
    def check(a):
        x = pow(a, d, n)
        if x == 1 or x == n - 1:
            return True
        for _ in range(s - 1):
            x = (x * x) % n
            if x == n - 1:
                return True
        return False
    bases = [2, 325, 9375, 28178, 450775, 9780504, 1795265022] # deterministic for bases
    return all(check(a % n) for a in bases)

def main():
    n_str = input("Enter a positive integer: ").strip()
    try:
        n = int(n_str)
    except ValueError:
        print("Invalid input. Please enter an integer.")
    return

if __name__ == "__main__":
    main()
```

> Used 1 reference
1. Naive trial division
• Idea: test divisibility by every integer $2 \dots \sqrt{n}$.
• Time: $O(\sqrt{n})$, Memory: $O(1)$. Deterministic.
• Use-case: tiny or educational.

2. Optimized trial division (skip even numbers / 6k+1 wheel)
• Idea: test only 2,3 then 6k+1 candidates.
• Time: $O(\sqrt{n}/3)$, Memory: $O(1)$. Deterministic.
• Use-case: single checks where code simplicity matters.

3. Wheel factorization (larger wheels: $2 \times 3 \times 5, \dots$)
• Idea: precompute residues coprime to small primes, skip many divisors.
• Time: reduces constant factors vs trial division. Deterministic.
• Use-case: faster trial division when checking many numbers but still small ranges.

4. Sieve of Eratosthenes
• Idea: mark multiples up to limit N to get all primes $\leq N$.
• Time: $O(N \log \log N)$, Memory: $O(N)$. Deterministic.
• Use-case: generate all primes up to N (best choice for bulk generation).

5. Segmented sieve
• Idea: apply sieve in segments to handle large N with limited memory.
• Time: same asymptotics as sieve. Memory: O(segment size + primes $\leq \sqrt{N}$).
• Use-case: generate primes in large intervals or when memory constrained.

6. Sieve of Atkin
• Idea: more complex sieve with some asymptotic constants better than Eratosthenes.
• Time: $O(N \log \log N)$ practically similar. Memory: $O(N)$. Deterministic.
• Use-case: niche; rarely worth complexity for most tasks.

7. Probabilistic tests — Miller-Rabin
• Idea: strong probable-prime test using modular exponentiation; repeat with random bases.
• Time: $O(k \log^3 n)$ per test (k bases). Memory: $O(1)$. Probabilistic. (error $\leq 4^{-k}$).

• Use-case: very large n primality checking where speed matters. Use a few bases for high confidence.
• Note: For 64-bit integers, a fixed small set of bases gives deterministic result.

8. Baillie-PSW

Add Context... lab1-4.py X
Add context (#), extensions (@), commands ()
Ask v GPT-5 mini v
Ln 56, Col 11 Spaces: 4 UTF-8 Python Python 3.13.7 Go Live BLACKBOXAI: Open Chat

EXECUTION FLOW

WITH FUNCTION:

- Main flow → `is_prime()` call → returns boolean
- Function scope isolated from main logic
- Single code path, reusable across program

WITHOUT FUNCTION:

- Inline logic directly in main flow
- All conditions evaluated in-place
- Must repeat code if used multiple times

TIME COMPLEXITY

Both approaches: $O(\sqrt{n})$	
• Check divisors from 2 to \sqrt{n}	
• Skip even numbers (optimization)	
• Early exit on first divisor found	
Operations count (approx):	
• $n = 100$: ~5 iterations worst case	
• $n = 1,000$: ~16 iterations	
• $n = 1,000,000$: ~500 iterations	

└─ PERFORMANCE FOR LARGE INPUTS ───────────	
WITH FUNCTION:	
✓ Identical algorithm performance	
+ Function call overhead: ~1-2 microseconds	
+ Negligible for single calls	
+ Better for multiple calls (code reuse)	
WITHOUT FUNCTION:	
✓ Slightly faster (no function call overhead)	
- Marginal difference: <1% faster	
- Code duplication increases file size	
- Harder to optimize if needed later	

└─ WHEN EACH APPROACH IS APPROPRIATE ───────────	
USE FUNCTION-BASED WHEN:	
✓ Checking primality multiple times in code	
✓ Building larger programs/libraries	
✓ Need unit testing capability	
✓ Want clean, maintainable code	

- | ✓ Extending logic later (e.g., caching) |
- | ✓ Working in teams (better code organization) |
- |
- | USE INLINE (NO FUNCTION) WHEN:
- | ✓ One-time check in simple script |
- | ✓ Performance-critical (negligible difference) |
- | ✓ Learning/prototyping phase |
- | ✓ Ultra-minimal code footprint required |
- | ⚠ Generally NOT recommended for production |
