Sieve of Eratosthenes

The Sieve of Eratosthenes is an ancient algorithm used to find all prime numbers up to a given limit.

- **Time Complexity:** *O(n log log n)*
- **Space Complexity:** *O(n)*
- **Edge Cases:** *When n < 2, the algorithm returns an empty list as there are no prime numbers.*

Algorithm Steps

1. *Create a list of consecutive integers from 2 to n.*
2. *Start with the first number (2), which is prime.*
3. *Remove all multiples of 2 from the list.*
4. *Move to the next number not removed (3), which is prime.*
5. *Remove all multiples of 3 from the list.*
6. *Repeat the process until no more numbers are left.*

Python Implementation

```
def sieve_of_eratosthenes(n):
    primes = [True] * (n + 1)
    primes[0] = primes[1] = False
    for i in range(2, int(n**0.5) + 1):
        if primes[i]:
            for j in range(i * i, n + 1, i):
                primes[j] = False
    return [i for i, is_prime in enumerate(primes) if is_prime]
```

Example Usage

```
print(sieve_of_eratosthenes(30)) # Output: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Written By **Krishna Singh**

File I/O

File I/O (Input/Output) operations allow reading from and writing to files.

File Operations

Opening a File

Use the open() function.

- ***Syntax:*** *file = open("filename", "mode")*
- ***Modes:*** *"r" (read), "w" (write), "a" (append), "b" (binary).*

Reading from a File:

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
```

Writing to a File:

```
with open("example.txt", "w") as file:
    file.write("Hello, World!")
```

Binary File Operations:

Use "rb" for reading and "wb" for writing binary files.

Closing a File

Use the close() method or the with statement (recommended).

Exceptions and Assertions

Exceptions handle runtime errors gracefully while assertions check conditions during code execution.

Exceptions

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
finally:
    print("Execution complete.")
```

Assertions

```
x = 5
assert x > 0, "x must be positive"
```

Modules

Modules are Python files containing reusable code.

Importing Modules:

```
import math
print(math.sqrt(16)) # Output: 4.0
```

Creating a Module:

```
my_module.py
def greet(name):
    return f"Hello, {name}!"
main.py
import my_module
print(my_module.greet("Alice"))
```

Abstract Data Types (ADTs)

ADTs define a data type in terms of its behaviour, not its implementation.

Python Implementation of Stack

```
class Stack:
    def __init__(self):
        self.items = []
    def push(self, item):
        self.items.append(item)
    def pop(self):
        return self.items.pop()
    def is_empty(self):
        return len(self.items) == 0
```

Example Usage

```
stack = Stack()
stack.push(1)
stack.push(2)
print(stack.pop())
# Output: 2
```

Classes

Classes are blueprints for creating objects in Python.

Class Definition

```
class Dog:
    def __init__(self, name):
        self.name = name
    def bark(self):
        return f"{self.name} says woof!"
```

Special Methods

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __str__(self):
        return f"Point({self.x}, {self.y})"
    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)
```

Example Usage

```
p1 = Point(1, 2)
p2 = Point(3, 4)
print(p1 + p2) # Output: Point(4, 6)
```

Inheritance

Inheritance promotes code reuse and modularity.

```
class Animal:
    def speak(self):
        return "Animal sound"

class Dog(Animal):
    def speak(self):
        return "Woof!"

dog = Dog()
print(dog.speak()) # Output: Woof!
```

Encapsulation

```
class Account:
    def __init__(self, balance):
        self.__balance = balance

    def get_balance(self):
        return self.__balance

    def set_balance(self, amount):
        if amount >= 0:
            self.__balance = amount
```

Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common superclass.

```
class Cat(Animal):
    def speak(self):
        return "Meow!"


for animal in (Dog(), Cat()):
    print(animal.speak())
```

Object-Oriented Programming (OOP)

OOP is a programming paradigm based on the concept of "objects," which can contain data and code to manipulate that data.

Key Principles

Encapsulation: *Bundling data with methods that operate on that data.*

Abstraction: *Hiding complex implementation details to focus on essential features.*

Inheritance: *Creating new classes from existing ones to promote code reusability.*

Polymorphism: *Allowing objects to be treated as instances of their parent class, enabling flexibility in code.*

Benefits

- *Improved code reusability and modularity*
- *Ease of maintenance and scalability*
- *Enhanced productivity and flexibility in development*

Example of inheritance, polymorphism, and encapsulation

```
class Animal:
    def speak(self):
        return "Animal sound"

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"

for animal in (Dog(), Cat()):
    print(animal.speak())
```

Output

```
Woof!
Meow!
```