

# Python for Discrete Mathematics

---

BRIDGING PROGRAMMING WITH  
MATHEMATICAL LOGIC

# Boolean Satisfiability



- Determine if there exists an assignment of truth values to variables that makes a given Boolean formula true.
- The most famous satisfiability problem is the CNF (Conjunctive Normal Form) satisfiability problem, where a formula is a conjunction of clauses, and each clause is a disjunction of literals.

---

## Problem Statement

Consider a simple SAT problem with the following Boolean expression in CNF:  
$$(A \vee B) \wedge (\neg A \vee C) \wedge (B \vee \neg C)$$

The task is to find an assignment of truth values to  $A$ ,  $B$ , and  $C$  that satisfies the expression, or determine that no such assignment exists.



## Solution Approach Using Python

We can approach this problem using a brute-force method by checking all possible truth assignments for *A*, *B*, and *C*.

```
def is_satisfiable():
    # Iterate over all possible truth values for A, B, and C
    for A in [True, False]:
        for B in [True, False]:
            for C in [True, False]:
                # Evaluate the CNF expression for the current truth
                # values
                if (A or B) and (not A or C) and (B or not C):
                    return True, A, B, C
            return False, None, None, None

# Check if the expression is satisfiable and print the result
satisfiable, A, B, C = is_satisfiable()
if satisfiable:
    print(f"Satisfiable with A={A}, B={B}, C={C}")
else:
    print("Not satisfiable")
```

## Explanation

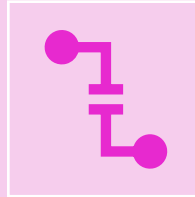
The function `is_satisfiable()` iterates over all possible combinations of truth values for A, B, and C.

For each combination, it evaluates the given Boolean expression.

If the expression evaluates to True for any combination of

A, B, and C, the function returns True along with the satisfying assignment.

If no satisfying assignment is found, the function returns False.



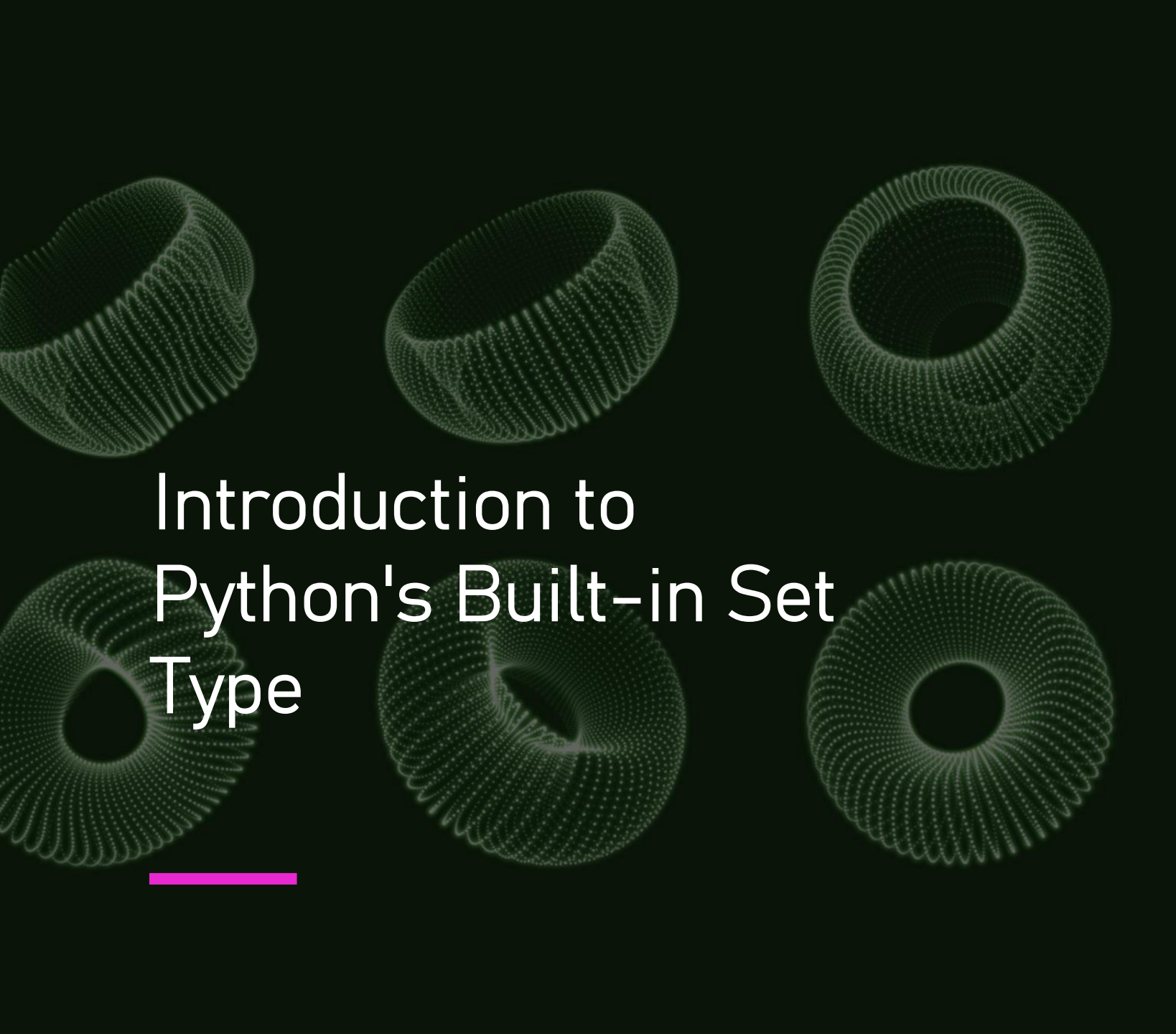
This brute-force approach is not scalable for larger SAT problems due to the exponential growth of possible assignments with the number of variables. For more complex problems, leveraging dedicated SAT solvers or implementing more sophisticated algorithms is recommended.



This example demonstrates a basic approach to solving a satisfiability problem in Python, providing insight into how logical expressions and satisfiability checks can be implemented and evaluated programmatically.

# Set Theory in Python





# Introduction to Python's Built-in Set Type

Python's built-in set type is an unordered collection of distinct hashable objects. It supports mathematical operations like union, intersection, difference, and symmetric difference, mirroring the set theory in mathematics.



# Relevance of Python's Set Type

---

**Efficiency:** Sets in Python are implemented using hash tables, which means that adding elements, checking for membership, and removing elements are operations that can be performed in constant time on average.

**Elimination of Duplicates:** Automatically removing duplicate entries simplifies data processing and analysis tasks.

**Mathematical Operations:** The ability to perform set operations like union, intersection, and difference directly on set objects makes it easier to implement algorithms and solutions that are based on set theory concepts.

**Flexibility:** Python sets can hold any hashable object, allowing for a wide range of applications, from simple collection processing to complex data analysis tasks.

**Use Cases:** From filtering data, performing fast membership tests, to implementing algorithms that require set operations, Python's set type is an incredibly powerful tool for both developers and data scientists.



## Defining Sets in Python

- Definition: A set is a collection of distinct (unique) objects.
- Python implementation: Using curly braces `{}` or the `set()` function.
- Sets automatically remove duplicates.

```
my_set = {1, 2, 3}
```

```
empty_set = set()
```

## Understanding Subsets

Definition: Set A is a subset of B if all elements of A are also elements of B.

Python implementation:  
The `<=` operator or  
`issubset()` method.

```
A = {1, 2}
```

```
B = {1, 2, 3}
```

```
print(A <= B) # True
```

```
print(A.issubset(B)) # True
```

## Union of Sets

- Definition: The union of sets A and B is a set of all elements from both A and B.

- Python implementation: The | operator or union() method.

```
A = {1, 2}
```

```
B = {2, 3}
```

```
print(A | B) # {1, 2, 3}
```

```
print(A.union(B)) # {1, 2, 3}
```

## Intersection of Sets

- Definition: The intersection of sets A and B is a set of elements that are common to both.
- Python implementation: The & operator or intersection() method.

```
A = {1, 2}
```

```
B = {2, 3}
```

```
print(A & B) # {2}
```

```
print(A.intersection(B)) # {2}
```

## Difference of Sets

•Definition: The difference between sets A and B ( $A-B$ ) is a set of elements that are in A but not in B.

•Python implementation: The `-` operator or `difference()` method.

```
A = {1, 2, 3}
```

```
B = {3, 4, 5}
```

```
print(A - B) # {1, 2}
```

```
print(A.difference(B)) # {1, 2}
```

## Complement of Sets

- Definition: The complement of set  $A$  refers to elements not in  $A$ , usually within a universal set  $U$ .
- Note: Python does not have a built-in set complement operation, but it can be simulated using the difference operation with a universal set.

$$U = \{1, 2, 3, 4, 5\}$$

$$A = \{1, 2, 3\}$$

$$\text{complement} = U - A$$

```
print(complement) # {4, 5}
```



## Calculating Set Cardinality Using Python

- Definition: Cardinality refers to the number of elements in a set.
- Python's len() function is for calculating the size of a set.

```
my_set = {1, 2, 3, 4, 5}
cardinality = len(my_set)
print(f"Cardinality is: {cardinality}")
```



## Cartesian Product Using Python Set Comprehension

Definition: Every element in set A is paired with every element in set B.

```
# Define two sets
```

```
A = {1, 2}
```

```
B = {3, 4}
```

```
# Compute Cartesian Product using set comprehension
```

```
cartesian_product = {(a, b) for a in A for b in B}
```

```
# Print result
```

```
print(cartesian_product)
```



---

# Quantifier Evaluation in Python

UNDERSTANDING UNIVERSAL AND  
EXISTENTIAL QUANTIFIERS USING PYTHON

# Understanding Quantifiers in Logic



Quantifiers express logical statements about elements in a domain.



$\forall$  (For All): A statement is true for all elements in the domain.



$\exists$  (There Exists): A statement is true for at least one element.



Python can be used to test the truth values of such statements.

---



## Defining the Predicate Function

```
#We define a predicate function P(x, y):
```

```
def P(x, y):
```

```
    return x + y == 0
```

```
# This function returns True if x and y sum  
to zero.
```

```
#It helps evaluate logical statements  
involving x and y.
```

---

# Approximating Real Numbers

- Since real numbers are infinite, we use an approximation:
- `real_numbers = range(-1000,1000)`
- `range` generates a finite set of numbers.
- We use this approximation to check logical statements.



# Evaluating Logical Statements

We check the truth values of the following statements:

1.  $\forall x \forall y P(x, y)$ : Is  $x + y = 0$  true for all  $x$  and  $y$ ?

2.  $\exists y \forall x P(x, y)$ : Is there a  $y$  such that for all  $x$ ,  $x + y = 0$ ?

3.  $\exists x \exists y P(x, y)$ : Does there exist at least one  $x$  and  $y$  such that  $x + y = 0$ ?

---





---

# Python Code to Evaluate Statements

- Using Python's `all()` and `any()` functions:
- `forall_x_forall_y = all(all(P(x, y) for y in real_numbers) for x in real_numbers)`
- `exists_y_forall_x = any(all(P(x, y) for x in real_numbers) for y in real_numbers)`
- `exists_x_exists_y = any(any(P(x, y) for y in real_numbers) for x in real_numbers)`





# Interpreting the Results

- The computed truth values are stored in a dictionary:

- `truth_values = {`
- `' $\forall x \forall y P(x, y)$ ': False,`
- `' $\exists y \forall x P(x, y)$ ': False,`
- `' $\exists x \exists y P(x, y)$ ': True`
- `}`

- · Not all  $x$  and  $y$  sum to zero.
- · There is no single  $y$  that satisfies all  $x$ .
- · At least one  $(x, y)$  pair satisfies the condition.



# Key Takeaways

- • Universal and existential quantifiers can be evaluated using Python.
- • We approximate real numbers to check logical statements.
- • Nested `all()` and `any()` functions simulate quantifiers.
- • The truth values confirm expected logical behavior.



# Sequences and Summations in Python

---

## Arithmetic Sequences in Python

- Definition and formula of an arithmetic sequence

$$a_n = a_1 + (n-1)d.$$

- Python example: Generating the first 10 terms of an arithmetic sequence.

```
def arithmetic_sequence(a1, d, n):  
    return [a1+(i-1)*d for i in range(1, n+1)]
```

```
print(arithmetic_sequence(1, 2, 10))
```

## Geometric Sequences in Python

Definition and formula of a  
geometric sequence

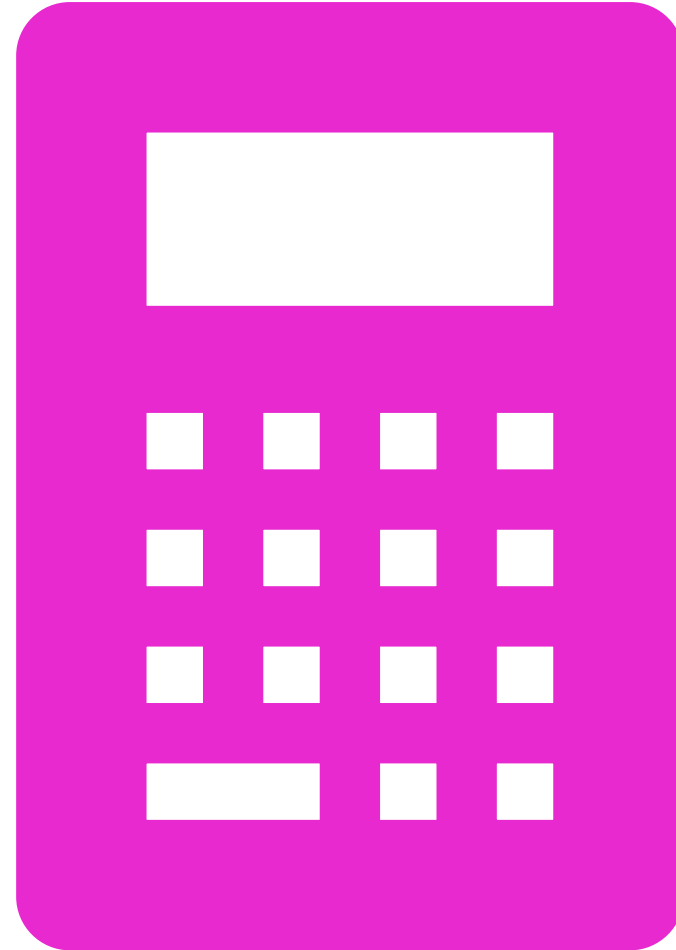
$$a_n = a_1 \cdot r^{n-1}$$

Python example: Calculating  
the first 5 terms of a  
geometric sequence.

```
def geometric_sequence(a1, r, n):  
    return [a1 * (r ** (i - 1)) for i in range(1, n+1)]  
  
print(geometric_sequence(1, 2, 5))
```

---

Calculate the sum of  
 $k^2$  from  $k=50$  to 100  
using a Python for loop





---

# Fibonacci Sequence

- Calculate the following sequence iteratively and recursively in Python (without looking it up!)
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

