Introduction to Python for Discrete Mathematics

BRIDGING PROGRAMMING WITH MATHEMATICAL LOGIC

Unlocking the Power of Discrete Mathematics with Python

- Logic
  - Boolean
  - Bitwise
  - Predicate
- Sets
- Functions
- Graphs
- Combinatorics
- Algorithms

Importance of Python:

SIMPLICITY AND READABILITY: PYTHON'S SYNTAX CLOSELY RESEMBLES STANDARD MATHEMATICAL NOTATION, MAKING IT AN IDEAL TOOL FOR DEMONSTRATING DISCRETE MATHEMATICS CONCEPTS.

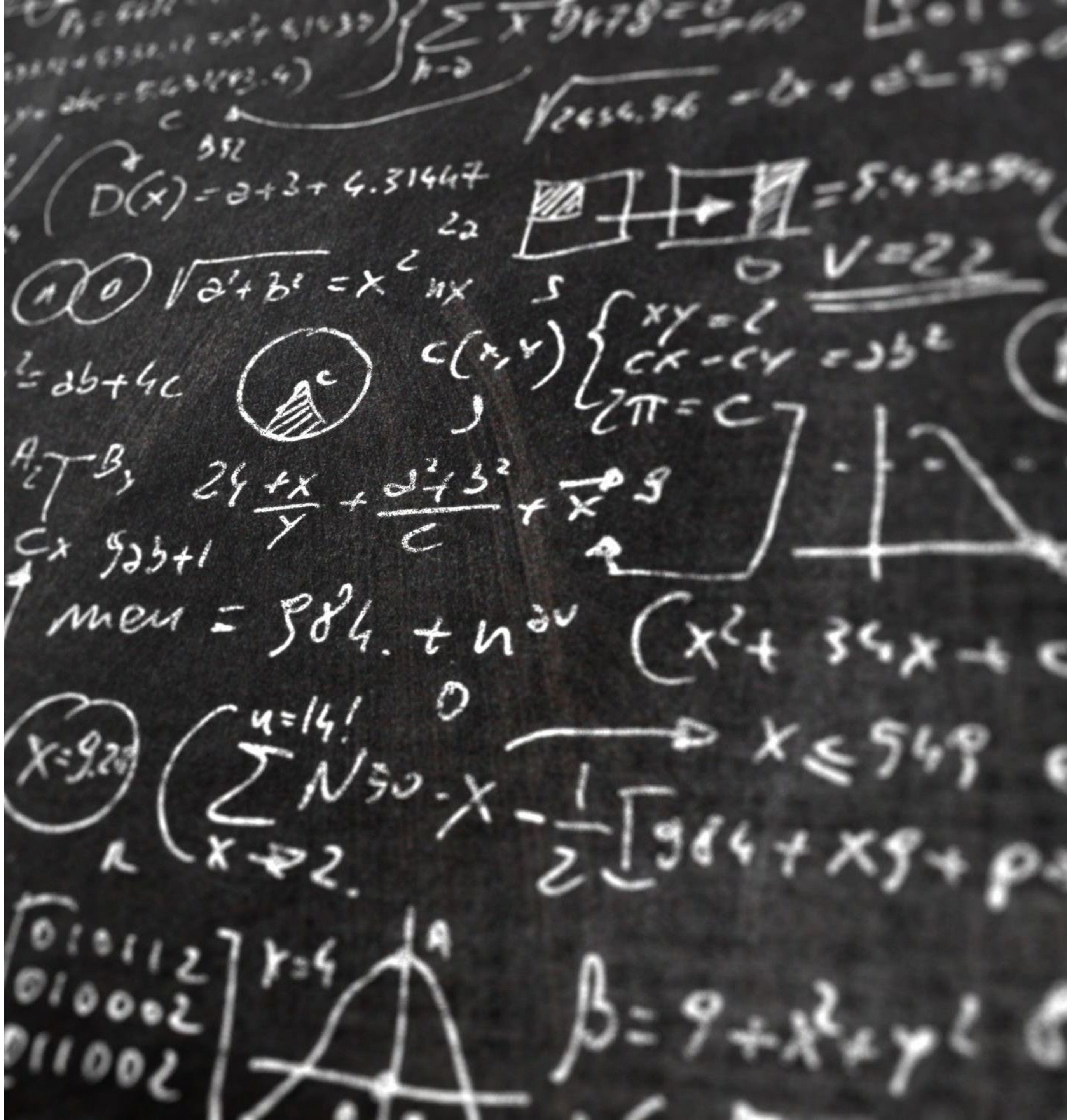
PRACTICAL APPLICATION OF DISCRETE MATH THEORIES THROUGH CODING, MAKING ABSTRACT CONCEPTS MORE TANGIBLE AND UNDERSTANDABLE.

Goals

•**Understand Basic Python Programming**: Familiarity with Python syntax and the ability to write simple programs.

•**Apply Python to Solve Discrete Math Problems**: Using Python to explore and solve problems in logic, set theory, graph theory, etc.

•**Develop Logical Thinking and Problem-Solving Skills**: Enhancing analytical skills through the lens of discrete mathematics and programming.



Python Data Types vs. Discrete Logic

- Bool (Boolean)
  - Represents True or False.
  - Used for logical operations and decision-making.
  - Discrete Logic Comparison: Corresponds to truth values in propositional logic.
- Int (Integer)
  - Represents whole numbers (positive, negative, zero).
  - No fractional part.
  - Discrete Logic Comparison: Aligns with the set of integers (Z) in mathematics.
- Float (Floating-Point Number)
  - Represents decimal numbers.
  - Subject to precision and rounding errors.
  - Discrete Logic Comparison: Approximates real numbers (R), useful in continuous math, not directly related to discrete logic but essential for numerical computations.

Python Strings

Basics of Strings

- Sequence of characters enclosed in quotes ('**hello**', "**world**").
- Immutable: once created, the content cannot be changed.

Creating Strings

- Single, double, or triple quotes for multi-line strings.
- Example: **my_string = "Hello, Python!"**

Accessing Characters

- Indexed from 0 for forward indexing, -1 for reverse.
- Example: **my_string[0]** (first character), **my_string[-1]** (last character).

Slicing Strings

- Extract substrings using : operator.
- Syntax: **[start:stop:step]**
- Example: **my_string[1:5]** extracts "**ello**".

String Operations

- Concatenation: **+** to join strings.
- Repetition: * for repeating strings.
- Example: **"Hello " + "World" → "Hello World"**, **"Hi" * 3 → "HiHiHi"**

Common String Methods

- **.upper()**, **.lower()**: Case conversion.
- **.strip()**: Removes whitespace from both ends.
- **.find()**, **.replace()**: Search and replace substrings.
- **.split()**: Splits string into a list based on a delimiter.

String Formatting

- Old style: **%** operator.
- **.format()** method: more versatile.
- F-strings (Python 3.6+): embed expressions **f"Hello, {name}!"**

Escape Sequences

- Use backslash **** for special characters: **\n** (newline), **\t** (tab).
- Example: **"First line.\nSecond line."**

Python Print

- Output to Console
- Comma separated or concatenated (+) strings
- print("Hello, world!")
- print("The answer is", 42)
- print("Sum: " + str(7 + 5))

Python input() function

Used to read a string from standard input (usually, the keyboard).

Pauses program execution and waits for the user to type something.

Once the user presses Enter, input() captures the input as a string.

Syntax: variable = input(prompt)

prompt (optional): A string presented to the user.

```
# string input
name = input("Enter your name: ")

# integer input
age = int(input("Enter your age: "))

# float input
distance=float(input("Enter distance"))
```

Problem: Greeting and Age Calculation

Objective: Write a Python script that does the following:

Instructions

Prompt the user to input their name using input().

Prompt the user to input their birth year and convert this input into an integer.

Calculate the age by subtracting the birth year from the current year.

Use the print() function to display a greeting that includes their name and age.

Problem: Greeting and Age Calculation

Objective: Write a Python script that does the following:

```
# Step 1: Ask for the user's name
name = input("What is your name? ")

# Step 2: Ask for the birth year and convert to integer
birth_year = int(input("What year were you born? "))

# Step 3: Calculate age
current_year = 2023
age = current_year – birth_year

# Step 4: Print the greeting message
print(f"Hello, {name}! You are {age} years old in
{current_year}.")
```

If Statements

Executes code based on an expression (condition) being true. Fundamental for decision-making in programming.

Condition: Any expression that evaluates to True or False. Can include comparisons, logical operations, etc.

Indentation

- Python uses indentation to define scope.
- Code block under if must be indented.

```
x = 10
if x > 0:
    print("x is positive")
elif x < 0:
    print("x is negative")
else:
    print("x is zero")
```

If Statement

Problem: Movie Ticket Pricing

Objective: Write a Python script to calculate the price of a movie ticket based on the age of the customer and whether it's a weekday or weekend. The pricing is as follows:

Children under 13: $10

Adults (13 to 59): $15 on weekdays, $20 on weekends

Seniors (60 and over): $12

Assumptions:

Assume "weekday" for Monday to Friday and "weekend" for Saturday and Sunday.

The user will input their age and the day of the week.

Instructions:

Prompt the user to input their age.

Prompt the user to input the day of the week (e.g., "Monday", "Tuesday", ... "Sunday").

Use if statements to determine the ticket price based on the user's age and the day of the week.

Print the ticket price.

If Statement

Problem: Movie Ticket Pricing

Objective: Write a Python script to calculate the price of a movie ticket based on the age of the customer and whether it's a weekday or weekend. The pricing is as follows:

Children under 13: $10

Adults (13 to 59): $15 on weekdays, $20 on weekends

Seniors (60 and over): $12

```
# Step 1: Ask for the user's age
age = int(input("Enter your age: "))

# Step 2: Ask for the day of the week
day = input("Enter the day of the week: ").lower()
# Convert to lowercase to handle mixed case inputs

# Step 3: Determine and print the ticket price
if age < 13:
    print("Ticket Price: $10")
elif age >= 13 and age <= 59:
    if day in ["saturday", "sunday"]:
        print("Ticket Price: $20")
    else:
        print("Ticket Price: $15")
else:  # Seniors
    print("Ticket Price: $12")
```

Python For Loops: Iterating with Ease

Purpose of For Loops

Used to repeat (iterate) over a sequence (such as string, list or set) and execute a block of code for each item.

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:  # for each loop
    print(fruit)

for i in range(len(fruits)): # for loop
    print(fuits[i])
```

Python For Loops: Problem: For each loop

Initialize a list of numbers.

Use a for loop to iterate through the list and calculate the sum.

Calculate the average by dividing the sum by the number of items in the list.

Print the results.

```
# Step 1: Initialize a list of numbers
numbers = [10, 20, 30, 40, 50]

# Step 2: Use a for loop to calculate the sum
total_sum = 0
for number in numbers:
    total_sum += number

# Step 3: Calculate the average
average = total_sum / len(numbers)

# Step 4: Print the sum and the average
print(f"Sum: {total_sum}, Average: {average}")
```

Python Functions

Blocks of reusable code designed to perform a specific task.

Parameters: Variables listed inside the parentheses in the function definition.

Arguments: Values passed to the function when it is called.

```
def function_name(parameters):
    # code block
function_name(arguments)


def add(a, b):
        return a + b
print(add(1,4))
```

Truth Tables in Python

Generating truth tables in Python involves systematically evaluating and displaying the truth values of logical expressions for all possible combinations of their inputs. Here's a step-by-step guide on how to create a function to generate truth tables for basic logical operators.

```
def truth_table_AND():
    print("A\tB\tA AND B")
    for A in [True, False]:
        for B in [True, False]:
            print(f"{A}\t{B}\t{A and B}")

def truth_table_OR():
    print("A\tB\tA OR B")
    for A in [True, False]:
        for B in [True, False]:
            print(f"{A}\t{B}\t{A or B}")
```

Truth Tables in Python

Custom Logical Expression

```
def truth_table_custom(expression):
    print("A\tB\tResult")
    for A in [True, False]:
        for B in [True, False]:
            result = eval(expression)
            print(f"{A}\t{B}\t{result}")

# Example usage:
expression = "not A and B or A and not B"
# Exclusive OR (XOR) simulation
truth_table_custom(expression)
```

Truth Tables in Python

Problem: Generate a truth table for a the expression (A AND B) OR C.

```
def generate_truth_table():
    print("A\tB\tC\t(A AND B) OR C")
    for A in [True, False]:
        for B in [True, False]:
            for C in [True, False]:
                result = (A and B) or C
                print(f"{A}\t{B}\t{C}\t{result}")

generate_truth_table()
```

Verifying Equivalence with Truth Tables

Consider the following two statements:

Statement 1:
A∧B (A AND B)

Statement 2:
¬(¬A∨¬B) (Not (Not A OR Not B))

These statements are logically equivalent based on De Morgan's Laws, which state that the negation of a disjunction is equivalent to the conjunction of the negations.

```
def verify_equivalence():
    print("A\tB\tA AND B\t¬(¬A OR ¬B)\tEquivalent")
    for A in [True, False]:
        for B in [True, False]:
            s1 = A and B
            s2 = not (not A or not B)
            equivalent = s1 == s2
            print(f"{A}\t{B}\t{s1}\t{s2}\t{equivalent}")
verify_equivalence()
```

Verifying Equivalence with a Function

Consider the following two statements:

Statement 1:
A∧B (A AND B)

Statement 2:
¬(¬A∨¬B) (Not (Not A OR Not B))

These statements are logically equivalent based on De Morgan's Laws, which state that the negation of a disjunction is equivalent to the conjunction of the negations.

```
def check_equivalence_all():
    # Initialize a flag to track equivalence across all cases
    all_equivalent = True

   # Iterate through all combinations of A and B
    for A in [True, False]:
        for B in [True, False]:
            s1 = A and B
            s2 = not (not A or not B)

             # Check if there's any case where they are not equivalent
            if s1 != s2:
                all_equivalent = False
                break  # No need to check further if any case is not equivalent

           if not all_equivalent:
                break  # Break the outer loop as well if any case is not equivalent

        # Print only if all cases are equivalent
     if all_equivalent:
        print("True")
    else:
        print("False")  # Optionally, indicate explicitly when not equivalent

check_equivalence_all()
```

Verifying Equivalence with a Function Simplified

Consider the following two statements:

Statement 1:
A∧B (A AND B)

Statement 2:
¬(¬A∨¬B) (Not (Not A OR Not B))

These statements are logically equivalent based on De Morgan's Laws, which state that the negation of a disjunction is equivalent to the conjunction of the negations.

```
def is_equivalent_for_all():
    # Iterate through all combinations of A and B
    for A in [True, False]:
        for B in [True, False]:
            s1 = A and B
            s2 = not (not A or not B)

             # Return False immediately if any case is not equivalent
             if s1 != s2:
                  return False

# If loop completes without returning, all cases are equivalent
     return True

# Call function and print its return value
result = is_equivalent_for_all()
print(result)
```

Bitwise Operations in Python

•Operate on binary representations of integers.

•Perform operations bit by bit over entire integer

AND (&): Sets each bit to 1 if both bits are 1.

OR (|): Sets each bit to 1 if one of two bits is 1.

XOR (^): Sets each bit to 1 if only one of two bits is 1.

NOT (~): Inverts all the bits.

Left Shift (<<): Shifts the bits to the left, filling with 0s.

Right Shift (>>): Shifts the bits to the right, discarding bits.

Bitwise Operations in Python

Examples

AND: a & b

5 & 3 (0b101 & 0b011) → 1 (0b001)

OR: a | b

5 | 3 (0b101 | 0b011) → 7 (0b111)

XOR: a ^ b

5 ^ 3 (0b101 ^ 0b011) → 6 (0b110)

NOT: ~a

~00000101 →11111010

Bitwise Operations in Python

Examples

```
# Initialize two integers
a = 5  # binary: 101
b = 3  # binary: 011

# Bitwise AND
result_and = a & b
print(f"AND Operation: {a} & {b} = {result_and}")  # Output
should be 1

# Bitwise OR
result_or = a | b
print(f"OR Operation: {a} | {b} = {result_or}")  # Output should
be 7

# Bitwise XOR
result_xor = a ^ b
print(f"XOR Operation: {a} ^ {b} = {result_xor}")  # Output
should be 6

# Bitwise NOT on 'a'
result_not = ~a
print(f"NOT Operation: ~{a} = {result_not}")
```

Understanding "There Exists" and "For All" in Python

"There Exists" (∃) in Logic

- Represents the existence of at least one element in a set that satisfies a given condition.
- Expressed as "∃x such that condition(x) is true."

"For All" (∀) in Logic

- Indicates that a condition holds for every element in a certain set.
- Expressed as "∀x, condition(x) is true."

```
# Example: Check if all numbers are even
numbers = [2, 4, 6, 8, 10]
all_even = True  # Assume all numbers are even initially
for number in numbers:
    if number % 2 != 0:  # If any number is not even
        all_even = False  # The assumption is proven false
        break  # No need to check further
print(f"All numbers are even: {all_even}")

# Example: Check if there exists an odd number
numbers = [2, 4, 5, 8, 10]
exists_odd = False  # Assume there are no odd numbers initially
for number in numbers:
    if number % 2 != 0:  # If an odd number is found
        exists_odd = True  # The assumption is proven true
        break  # No need to check further
print(f"There exists an odd number in the list: {exists_odd}")
```

Understanding "There Exists" and "For All" in Python using any and all

"There Exists" (∃) in Logic

- Represents the existence of at least one element in a set that satisfies a given condition.
- Expressed as "∃x such that condition(x) is true."

"For All" (∀) in Logic

- Indicates that a condition holds for every element in a certain set.
- Expressed as "∀x, condition(x) is true."

```
# Example: Check if there exists an odd number
numbers = [1, 2, 3, 4, 5]
exists_odd = any(number % 2 != 0 for number in numbers)
print(f"There exists an odd number in the list: {exists_odd}")

# Example: Check if all numbers are even
numbers = [2, 4, 6, 8, 10]
all_even = all(number % 2 == 0 for number in numbers)
print(f"All numbers are even: {all_even}")
```

Problem: Integer Pair Relationships

Given the set of integers from 1 to 10 for *x* and *y*, we want to investigate two compound statements simultaneously:

1. **There Exists** (∃) an *x* such that for **all** *y* in the set, $x \cdot y$ is even.
2. **For All** (∀) *y*, there **exists** an *x* such that $x+y$ is odd.

This problem explores the existence of a specific *x* that when multiplied by any *y* gives an even product, and for every *y*, there is some *x* that makes their sum odd.

```
# Range of integers
range_integers = range(1, 11)

# Condition 1: Check for an x that makes x * y even for all y
exists_x_for_all_y_even = any(all((x * y) % 2 == 0 for y in range_integers)
for x in range_integers)

# Condition 2: Check that for all y, there exists an x that makes x + y odd
all_y_exists_x_sum_odd = all(any((x + y) % 2 != 0 for x in range_integers)
for y in range_integers)

# Print results
print(f"Exists an x for all y, x*y is even: {exists_x_for_all_y_even}")
print(f"For all y, there exists an x, x+y is odd: {all_y_exists_x_sum_odd}")
```