A relation on a set A is a subset of the Cartesian product $A \times A$. It describes how elements of A are related to each other.

Types of Relations

Reflexive Relation: A relation R on A is reflexive if $(a, a) \in R$ for all $a \in A$.

Example: The "greater than or equal to" relation ($\geq$) on real numbers.

Symmetric Relation: A relation R on A is symmetric if $(a, b) \in R \Rightarrow (b, a) \in R$.

Example: The "is a sibling of" relation.

Antisymmetric Relation: A relation R on A is antisymmetric if $(a, b) \in R$ and $(b, a) \in R$ together imply $a = b$.

Example: The "less than or equal to" relation ($\leq$) on real numbers.

Transitive Relation: A relation R on A is transitive if $(a, b) \in R$ and $(b, c) \in R$ together imply $(a, c) \in R$.

Example: The "is an ancestor of" relation.

Equivalence Relation: A relation that is reflexive, symmetric, and transitive.

Example: The "has the same birthday as" relation.

Partial Order Relation: A relation that is reflexive, antisymmetric, and transitive.

Example: The "subset" relation ($\subseteq$) on a power set.

Total Order Relation: A partial order relation where every pair of elements is comparable.

Example: The "less than or equal to" ($\leq$) relation on real numbers.

Types of Relations with Mathematical Examples

Reflexive Relation: A relation R on A is reflexive if $(a, a) \in R$ for all $a \in A$.

Example: Let $A = \{1, 2, 3\}$. The relation

$$R = \{(1, 1), (2, 2), (3, 3), (1, 2), (2, 1)\}$$

is **reflexive**. A reflexive relation must have all pairs (a, a) for every $a \in A$. A reflexive example would be:

$$R = \{(1, 1), (2, 2), (3, 3), (1, 2), (2, 1), (2, 3), (3, 2)\}.$$

Symmetric Relation: A relation R on A is symmetric if $(a, b) \in R \Rightarrow (b, a) \in R$.

Example: Let $A = \{1, 2, 3\}$. The relation

$$R = \{(1, 2), (2, 1), (2, 3), (3, 2)\}$$

is symmetric because whenever $(a, b) \in R$, then $(b, a) \in R$ as well. A non-symmetric example would be:

$$R = \{(1, 2), (2, 3)\}$$

(because $(2, 1)$ and $(3, 2)$ are missing).

Antisymmetric Relation: A relation R on A is antisymmetric if $(a, b) \in R$ and $(b, a) \in R$ together imply $a = b$.

Example: Let $A = \{1, 2, 3\}$. The relation

$$R = \{(1, 1), (2, 2), (3, 3), (1, 2)\}$$

is antisymmetric because if $(a, b) \in R$ and $(b, a) \in R$, then $a = b$. A non-antisymmetric example would be:

$$R = \{(1, 2), (2, 1)\}$$

(because $1 \neq 2$ and both $(1, 2)$ and $(2, 1)$ are present).

Transitive Relation: A relation R on A is transitive if $(a, b) \in R$ and $(b, c) \in R$ together imply $(a, c) \in R$.

Example: Let $A = \{1, 2, 3\}$. The relation

$$R = \{(1, 2), (2, 3), (1, 3)\}$$

is transitive because $(1, 2) \in R$ and $(2, 3) \in R$ imply $(1, 3) \in R$. A non-transitive example would be:

$$R = \{(1, 2), (2, 3)\}$$

(because $(1, 3)$ is missing).

Equivalence Relation: A relation that is reflexive, symmetric, and transitive.

Example: Let $A = \{1, 2, 3, 4\}$. The relation

$$R = \{(1, 1), (2, 2), (3, 3), (4, 4), (1, 3), (3, 1)\}$$

is an equivalence relation because it is reflexive, symmetric, and transitive.

Partial Order Relation: A relation that is reflexive, antisymmetric, and transitive.

Example: Consider the "divides" relation on $A = \{1, 2, 4, 8\}$:

$$R = \{(1, 1), (2, 2), (4, 4), (8, 8), (1, 2), (2, 4), (4, 8), (1, 4), (1, 8), (2, 8)\}.$$

This relation is reflexive, antisymmetric, and transitive.

Total Order Relation: A partial order relation where every pair of elements is comparable.

Example: Consider the "less than or equal to" ($\leq$) relation on $A = \{1, 2, 3\}$:

$$R = \{(1, 1), (2, 2), (3, 3), (1, 2), (2, 3), (1, 3)\}.$$

Every element can be compared with every other element, making it a total order.

Non-Total Order Example:

If we define the "divisibility" relation on $A = \{2, 3, 6\}$:

$$R = \{(2, 2), (3, 3), (6, 6), (2, 6), (3, 6)\}$$

It is **not total** because 2 and 3 are not comparable (neither 2 divides 3 nor 3 divides 2). Hence, this is a **Partial Order** but not a **Total Order**.

Python Implementation

a) ITER is conducting course registration for the new semester. Each student can enroll in any of the available subjects. Given a list of students and a list of subjects, generate all possible student-subject enrollment pairs.

- Students: Asutosh, Ayushman, Mohit, Priya
- Subjects: Mathematics, Physics, Computer Science, Economics

Student-Subject Enrollment Pairs

Let S be the set of students and T be the set of subjects:

$$S = \{\text{Asutosh}, \text{Ayushman}, \text{Mohit}, \text{Priya}\}$$

$$T = \{\text{Mathematics}, \text{Physics}, \text{Computer Science}, \text{Economics}\}$$

The Cartesian product $S \times T$ gives all possible enrollment pairs:

$$S \times T = \begin{cases} (\text{Asutosh}, \text{Mathematics}), & (\text{Asutosh}, \text{Physics}), & (\text{Asutosh}, \text{Computer Science}), & (\text{Asutosh}, \text{Ec} \\ (\text{Ayushman}, \text{Mathematics}), & (\text{Ayushman}, \text{Physics}), & (\text{Ayushman}, \text{Computer Science}), & (\text{Ayushman}, \text{E} \\ (\text{Mohit}, \text{Mathematics}), & (\text{Mohit}, \text{Physics}), & (\text{Mohit}, \text{Computer Science}), & (\text{Mohit}, \text{Eco} \\ (\text{Priya}, \text{Mathematics}), & (\text{Priya}, \text{Physics}), & (\text{Priya}, \text{Computer Science}), & (\text{Priya}, \text{Eco} \end{cases}$$

Python Code for Student-Subject Enrollment Pairs

The following Python program generates all possible student-subject enrollment pairs using the Cartesian product:

```
from itertools import product

# List of students and subjects
students = {"Asutosh", "Ayushman", "Mohit", "Priya"}
subjects = {"Mathematics", "Physics", "Computer␣Science", "Economics"}

# Generate all possible enrollment pairs using Cartesian product
enrollments = set(product(students, subjects))

# Display the enrollment pairs
for student, subject in enrollments:
    print(f"({student},␣{subject})")
```

b) In a social media app, friendships are symmetric: If A is a friend of B, then B must also be a friend of A. Define a Python function to verify if a given friendship relation is symmetric and complete the missing pairs if necessary. Given two relations are:

- $R1$: {("Anurag", "Nitish"), ("Priyabrata", "Koustav"), ("Prateek", "Asutosh")}
- $R2$: {("Priya", "Nikita"), ("Nikita", "Priya"), ("Sancheeta", "Shreyashree"),("Shreyashree", "Sancheeta"

```
def make_symmetric(relation):
    """
    Ensures that the given friendship relation is symmetric.

    Args:
        relation (set): A set of tuples representing friendship pairs.

    Returns:
        set: A symmetric version of the given relation.
    """
    symmetric_relation = set(relation)  # Copy the given relation

    for a, b in relation:
        if (b, a) not in symmetric_relation:  # Check if reverse pair exists
            symmetric_relation.add((b, a))   # Add missing pair

    return symmetric_relation

# Given friendship relations
R1 = {("Anurag", "Nitish"), ("Priyabrata", "Koustav"), ("Prateek", "Asutosh")
R2 = {("Priya", "Nikita"), ("Nikita", "Priya"), ("Sancheeta", "Shreyashree"),

# Ensure symmetry
R1_symmetric = make_symmetric(R1)
R2_symmetric = make_symmetric(R2)

# Display results
print("Symmetric␣R1:", R1_symmetric)
print("Symmetric␣R2:", R2_symmetric)
```

c) A flight network records direct flights between cities. If there is a flight from A to B and from B to C, then there should be a direct or indirect flight from A to C (transitive relation). Check if the relation is transitive.

- *flights* = {("New York", "London"), ("London", "Paris"), ("Paris", "Berlin"), ("New York", "Paris")}

```
def is_transitive(relation):
    """
    Checks if a given relation is transitive.

    Args:
        relation (set): A set of tuples representing direct flights.

    Returns:
        bool: True if transitive, False otherwise.
    """
    for (A, B) in relation:
        for (X, C) in relation:
            if B == X and (A, C) not in relation:
                return False  # Missing transitive flight
    return True

# Given flight network
flights = {("New York", "London"), ("London", "Paris"),
           ("Paris", "Berlin"), ("New York", "Paris")}

# Check transitivity
if is_transitive(flights):
    print("The flight network is transitive.")
else:
    print("The flight network is NOT transitive.")
```

Definition of a Function in Mathematics

A function f from a set A to a set B is a rule that assigns each element of A exactly one element of B. It is denoted as:

$$f : A \to B$$

where $f(a) = b$ for $a \in A$ and $b \in B$.

Types of Functions

1. Injective (One-to-One) Function

A function $f : A \to B$ is called **injective** if different elements in A map to different elements in B. Mathematically,

$$f(a_1) = f(a_2) \Rightarrow a_1 = a_2, \quad \forall a_1, a_2 \in A$$

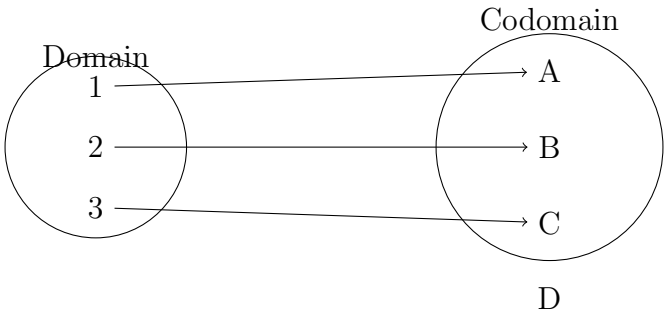
Example: The function $f(x) = 2x$ (for $x \in \mathbb{R}$) is injective because different x-values give different function values.
Consider a function:

$$f : \{1, 2, 3\} \to \{A, B, C, D\}$$

defined as:

$$f(1) = A, \quad f(2) = B, \quad f(3) = C$$

Each element in the domain has a unique image in the codomain, ensuring **injectivity** (one-to-one).



2. Surjective (Onto) Function

A function $f : A \to B$ is called **surjective** if every element of B has at least one pre-image in A. Mathematically,

$$\forall b \in B, \exists a \in A \text{ such that } f(a) = b$$

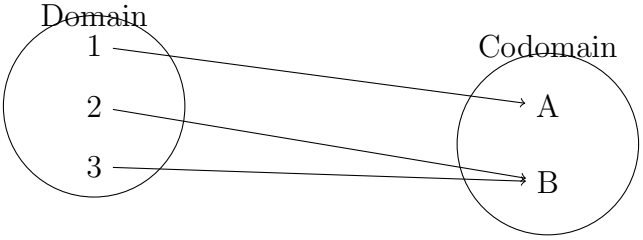
Example: The function $f(x) = x^2$ from $\mathbb{R}$ to $\mathbb{R}_{\geq 0}$ is surjective, as every non-negative real number has a pre-image.
Consider a function:

$$f : \{1, 2, 3\} \to \{A, B\}$$

defined as:

$$f(1) = A, \quad f(2) = B, \quad f(3) = B$$

Since every element in the codomain $\{A, B\}$ is mapped by at least one element from the domain $\{1, 2, 3\}$, the function is **surjective** (onto).



3. Bijective (One-to-One and Onto) Function

A function is **bijective** if it is both injective and surjective. This means every element of B has exactly one pre-image in A. A function is **bijective** if it is both injective (one-to-one) and surjective (onto).

Consider the function:

$$f : \{1, 2, 3\} \to \{A, B, C\}$$

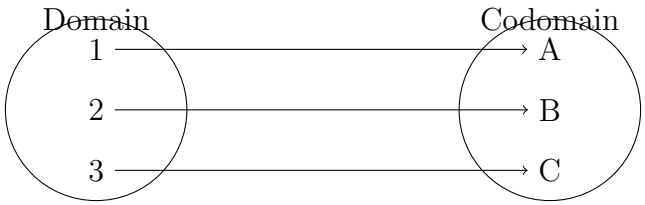
defined as:

$$f(1) = A, \quad f(2) = B, \quad f(3) = C$$

Since:

- Each element in the **domain** has a unique image in the **codomain** (**injectivity**).
- Every element in the **codomain** is mapped by exactly one element from the **domain** (**surjectivity**).

The function is **bijective**.



Example: The function $f(x) = x + 3$ (for $x \in \mathbb{R}$) is bijective since it is both one-to-one and onto.

Python Program to Check Injectivity of a Function

The following Python program checks whether a given function is injective (one-to-one).

```
def is_injective(function, domain):
    """
    Checks if a given function is injective (one-to-one).

    Parameters:
    function : callable
        The function to check.
    domain : list or set
        The set of input values.

    Returns:
    bool : True if the function is injective, False otherwise.
    """
    output_values = {}

    for x in domain:
        fx = function(x)
        if fx in output_values:
            return False  # Found duplicate output, function is not injective
        output_values[fx] = x  # Store the output in a dictionary

    return True  # No duplicate outputs found

# Example functions
def f1(x):
    return 2*x  # Injective function

def f2(x):
    return x**2  # Not injective in real numbers

# Test cases
domain1 = list(range(-10, 11))  # Checking from -10 to 10
print("f1(x) = 2x is injective:", is_injective(f1, domain1))  # Should return
print("f2(x) = x^2 is injective:", is_injective(f2, domain1))  # Should retur
```

Python Program to Check Surjectivity of a Function

The following Python program checks whether a given function is surjective (onto) for a specified domain and codomain.

```
def is_surjective(function, domain, codomain):
    """
```

```
    Checks if a given function is surjective (onto).

    Parameters:
    function : callable
        The function to check.
    domain : list or set
        The set of input values (domain).
    codomain : list or set
        The set of expected output values (codomain).

    Returns:
    bool : True if the function is surjective, False otherwise.
    """
    output_values = {function(x) for x in domain}  # Compute function outputs

    return output_values >= set(codomain)  # Check if codomain is covered

# Example functions
def f1(x):
    return x - 3  # Surjective for domain = R, codomain = R

def f2(x):
    return x**2  # Not surjective for R to R

# Test cases
domain1 = list(range(-10, 11))  # Checking for x in [-10, 10]
codomain1 = list(range(-13, 8))  # Example codomain

print("f1(x) = x - 3 is surjective:", is_surjective(f1, domain1, codomain1))
# Should return True
print("f2(x) = x^2 is surjective:", is_surjective(f2, domain1, codomain1))
# Should return False
```

Key differences between function and relation:

- Every function is a relation, but not every relation is a function.
- Functions have a strict rule: one input → one output, while relations allow multiple outputs for the same input.

Key Differences Between Mathematical Functions and Python Functions

- **Definition:**
  - Mathematical: Maps elements from a domain to a codomain.
  - Python: A block of code that computes and returns a result.
- **Input/Output:**
  - Mathematical: Exactly one output per input.
  - Python: Can return multiple outputs or none (`None`).

- **Notation:**
  - Mathematical: $f(x) = \ldots$ or $f : X \to Y$.
  - Python: Defined using `def`, e.g., `def f(x): ...`.
- **Execution:**
  - Mathematical: Abstract and conceptual.
  - Python: Executable code running on a computer.

Mathematical Logics

Simplification of Logical Expression

1. Disjunctive Normal Form (DNF)

Definition: A Boolean expression is in **Disjunctive Normal Form (DNF)** if it is written as a **disjunction (OR, $\vee$) of conjunctions (AND, $\wedge$)** of literals.

General Structure:

$$(A \wedge B) \vee (B \wedge C) \vee (\neg A \wedge C)$$

Example:

$$(A \wedge B) \vee (\neg B \wedge C)$$

Key Properties:

- Each term is a **minterm** (AND of literals).
- The whole expression is **ORed together**.
- Useful in decision trees and pattern recognition.

2. Conjunctive Normal Form (CNF)

Definition: A Boolean expression is in **Conjunctive Normal Form (CNF)** if it is written as a **conjunction (AND, $\wedge$) of disjunctions (OR, $\vee$)** of literals.

General Structure:

$$(A \vee B) \wedge (B \vee C) \wedge (\neg A \vee C)$$

Example:

$$(A \vee \neg B) \wedge (B \vee C)$$

Key Properties:

- Each term is a **clause** (OR of literals).
- The whole expression is **ANDed together**.
- CNF is useful for SAT solvers and logic programming.

3. Example: CNF vs DNF

Given a Boolean function:

$$F(A, B, C) = (\neg A \wedge B) \vee (\neg(A \vee C)) \vee (\neg B \wedge C)$$

Using simplification techniques:

- **DNF**: $\neg A \vee (\neg B \wedge C)$
- **CNF**: $(\neg A \vee \neg B \vee C)$

4. Key Differences Between CNF and DNF

Feature	CNF (AND of ORs)	DNF (OR of ANDs)
Structure	Clauses with OR ($\vee$) connected by AND ($\wedge$)	Minterms with AND ($\wedge$) connected by OR ($\vee$
SAT Solvers	Preferred	Not preferred
Decision Trees	Not preferred	Preferred
Example	$(A \vee B) \wedge (C \vee D)$	$(A \wedge B) \vee (C \wedge D)$
Conversion	Distribute OR over AND	Distribute AND over OR

Equivalent Expressions for XOR

1. **Standard Definition:**
$$A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)$$
2. **Using OR and AND:**
$$A \oplus B = (A \vee B) \wedge (\neg A \vee \neg B)$$

Conditional (Implication):

1. The implication $A \to B$ (If A, then B) can be expressed as:
$$A \to B = \neg A \vee B$$

Biconditional (If and Only If):

1. The biconditional $A \leftrightarrow B$ (A if and only if B) can be expressed as:
$$A \leftrightarrow B = (A \wedge B) \vee (\neg A \wedge \neg B)$$
or equivalently,
$$A \leftrightarrow B = (A \vee \neg B) \wedge (\neg A \vee B)$$

Given Expression:
$$F(A, B, C) = (\neg A \wedge B) \vee (\neg(A \vee C)) \vee (\neg B \wedge C)$$

Step 1: Apply De Morgan's Theorem
$$\neg(A \vee C) = \neg A \wedge \neg C$$
So, the expression becomes:
$$F = (\neg A \wedge B) \vee (\neg A \wedge \neg C) \vee (\neg B \wedge C)$$

Step 2: Distribute Terms
$$F = \neg A \wedge (B \vee \neg C) \vee (\neg B \wedge C)$$

Step 3: Distributive rule
$$F = \neg A \wedge 1[use(B \vee \neg C) \vee (\neg B \wedge C) = 1]$$

Final Simplified Expression:
$$F(A, B, C) = \neg A$$

Complex Expression	Simplified Equivalent
$F(A, B, C) = (\neg A \land B) \lor (\neg(A \lor C)) \lor (\neg B \land C)$	$\neg A$
$G(A, B, C) = (A \oplus B) \lor (B \Rightarrow C) \land (\neg A \lor C)$	$\neg A \land \neg B \lor C$
$H(A, B, C) = (A \text{ NAND } B) \land (\neg A \lor B) \land (A \lor \neg B \lor C)$	Minimal Form (No further simplification)
$M(A, B, C) = (A \land B) \lor (B \land C) \lor (A \land C)$	Minimal Form (No further simplification)
$J(A, B, C) = \neg(\neg A \lor \neg(\neg B \lor C)) \lor (A \land B)$	Minimal Form (No further simplification)

Table 1: Complex Logical Expressions and Their Simplified Forms

Disjunctive Normal Form (DNF)
$(A \land B) \lor (A \land C)$
$(A \land B) \lor (\neg A \land C)$
$(A \land B \land C) \lor (\neg A \land \neg B)$
$(A \land B) \lor (A \land \neg C) \lor (\neg A \land C)$
$(A \land B \land C) \lor (A \land \neg B \land D) \lor (\neg A \land C)$

Conjunctive Normal Form (CNF)
$(A \lor C) \land (B \lor C)$
$(A \lor C) \land (\neg A \lor B)$
$(A \lor \neg A) \land (A \lor \neg B) \land (B \lor \neg A) \land (B \lor \neg B) \land (C \lor \neg A) \land (C \lor \neg B)$
$(A \lor C) \land (B \lor C) \land (A \lor \neg C)$
$(A \lor C) \land (A \lor \neg B \lor D) \land (B \lor C) \land (B \lor \neg B \lor D) \land (C \lor \neg B \lor D)$

Introduction

This document presents a Python script that converts a Boolean expression from **Disjunctive Normal Form (DNF)** to **Conjunctive Normal Form (CNF)** using the `sympy` library.

1 Python Code for DNF to CNF

The following Python script demonstrates the conversion process:

Alternatively, the inline Python code is:

```
from sympy import symbols
from sympy.logic.boolalg import Or, And, to_cnf, to_dnf

""" Define Boolean variables"""
A, B, C, D = symbols('A␣B␣C␣D')


dnf_expr = Or(And(A, B), And(C, D))

"""Convert DNF to CNF"""
cnf_expr = to_cnf(dnf_expr, simplify=True)

"""Convert CNF back to DNF"""
dnf_converted = to_dnf(cnf_expr, simplify=True)

"""Print results"""
print("DNF␣Expression:", dnf_expr)
print("CNF␣Expression:", cnf_expr)
```

```
print("Converted␣DNF␣Expression:", dnf_converted)
```

2 Python Code to simplify a complex logical expression using sympy

```
from sympy import symbols
from sympy.logic.boolalg import simplify_logic

A, B, C = symbols('A␣B␣C')

expr = (A & B) | (A & ~B)  """ A AND B OR A AND NOT B"""
simplified_expr = simplify_logic(expr)

print("Original␣Expression:", expr)
print("Simplified␣Expression:", simplified_expr)
```

3 Python Code to print a truth table of a complex logical expression using sympy

The following Python script generates a truth table for a given Boolean expression:

```
from sympy import symbols
from sympy.logic.boolalg import And, Or, Not
from sympy.logic.boolalg import truth_table

# Define Boolean variables
A, B, C = symbols('A␣B␣C')
# Generate and print the truth table
print("Truth␣Table␣for:", expr)
print("A␣|␣B␣|␣C␣|␣Output")
print("---------------------")
for values, result in truth_table(expr, [A, B, C]):
   print(f"{int(values[0])}|{int(values[1])}|{int(values[2])}|{int(bool(resul
```

4 Python Code to print a truth table of a complex logical expression without using sympy

```
from itertools import product

# Define Boolean variables
variables = ['A', 'B', 'C']

# Define the logical expression manually
def logical_expression(A, B, C):
    return (A and B) or not C

# Generate and print the truth table
print("A␣|␣B␣|␣C␣|␣Output")
print("---------------------")
```

```
# Iterate over all possible truth values (0 and 1) for A, B, C
for values in product([0, 1], repeat=len(variables)):
    A, B, C = values
    result = logical_expression(A, B, C)
    print(f"{A} | {B} | {C} | {int(result)}")
```

5 Python Code to check equivalence of two expressions using sympy

```
from sympy import symbols
from sympy.logic.boolalg import Equivalent

# Define Boolean variables
A, B, C, D = symbols('A B C D')

# Define two logical expressions
expr1 =
expr2 =

# Check equivalence using Equivalent
equivalence = Equivalent(expr1, expr2)

# Print result
if equivalence == True:
    print("The two expressions are equivalent.")
else:
    print("The two expressions are NOT equivalent.")

# Print the equivalence expression
print("Equivalence Expression:", equivalence)
```

Python provides built-in functions for converting numbers between different number systems. Below are the methods along with examples.

Decimal to Other Bases

- **Binary:** `bin(n)` → Converts decimal to binary (prefix: `'0b'`)

  ```
  print(bin(10))  # Output: '0b1010'
  ```

- **Octal:** `oct(n)` → Converts decimal to octal (prefix: `'0o'`)

  ```
  print(oct(10))  # Output: '0o12'
  ```

- **Hexadecimal:** `hex(n)` → Converts decimal to hexadecimal (prefix: `'0x'`)

  ```
  print(hex(255))  # Output: '0xff'
  ```

Other Bases to Decimal

- **Binary to Decimal:** `int(binary`$_s tr, 2$)

- **Octal to Decimal:** `int(octal_str, 8)`

```
print(int('12', 8))  # Output: 10
```

- **Hexadecimal to Decimal:** `int(hex_str, 16)`

```
print(int('ff', 16))  # Output: 255
```

Conversions Between Binary, Octal, and Hexadecimal

Binary to Octal/Hexadecimal

- **Binary to Octal:**

```
print(oct(int('1010', 2)))  # Output: '0o12'
```

- **Binary to Hexadecimal:**

```
print(hex(int('1010', 2)))  # Output: '0xa'
```

Octal to Binary/Hexadecimal

- **Octal to Binary:**

```
print(bin(int('12', 8)))  # Output: '0b1010'
```

- **Octal to Hexadecimal:**

```
print(hex(int('12', 8)))  # Output: '0xa'
```

Hexadecimal to Binary/Octal

- **Hexadecimal to Binary:**

```
print(bin(int('A', 16)))  # Output: '0b1010'
```

- **Hexadecimal to Octal:**

```
print(oct(int('A', 16)))  # Output: '0o12'
```

Conclusion

These functions make it easy to convert numbers between different bases in Python. Remember that the `int()` function is useful for converting strings of different bases into decimal, and `bin()`, `oct()`, and `hex()` help convert decimal numbers into binary, octal, and hexadecimal formats, respectively. Python provides methods for converting floating-point numbers between different number systems. Since floating-point numbers have both integer and fractional parts, special techniques are used to handle the fractional part.

6 Decimal to Other Bases

- **Decimal to Binary:**

```
def float_to_binary(num, precision=10):
integer_part = bin(int(num))
fractional_part = num - int(num)
binary_fraction = []

while fractional_part and len(binary_fraction) < precision:
    fractional_part *= 2
    bit = int(fractional_part)
    binary_fraction.append(str(bit))
    fractional_part -= bit

return f"{integer_part}.{''.join(binary_fraction)}"

print(float_to_binary(10.625))  # Output: '0b1010.101'
```

- **Decimal to Octal:**

```
def float_to_octal(num, precision=10):
integer_part = oct(int(num))
fractional_part = num - int(num)
octal_fraction = []

while fractional_part and len(octal_fraction) < precision:
    fractional_part *= 8
    digit = int(fractional_part)
    octal_fraction.append(str(digit))
    fractional_part -= digit

return f"{integer_part}.{''.join(octal_fraction)}"

print(float_to_octal(10.625))  # Output: '0o12.5'
```

- **Decimal to Hexadecimal:**

```
def float_to_hex(num, precision=10):
integer_part = hex(int(num))
fractional_part = num - int(num)
hex_fraction = []

while fractional_part and len(hex_fraction) < precision:
    fractional_part *= 16
    digit = int(fractional_part)
    hex_fraction.append(hex(digit)[2:])
    fractional_part -= digit

return f"{integer_part}.{''.join(hex_fraction)}"

print(float_to_hex(10.625))  # Output: '0xa.a'
```

7 Other Bases to Decimal

- **Binary to Decimal:**

```
def binary_to_decimal(binary_str):
integer_part, fractional_part = binary_str.split('.')
decimal_value = int(integer_part, 2)
for i, digit in enumerate(fractional_part):
decimal_value += int(digit) * (2 ** -(i + 1))
return decimal_value

print(binary_to_decimal('1010.101'))  # Output: 10.625
```

- **Octal to Decimal:**

```
def octal_to_decimal(octal_str):
integer_part, fractional_part = octal_str.split('.')
decimal_value = int(integer_part, 8)
for i, digit in enumerate(fractional_part):
decimal_value += int(digit) * (8 ** -(i + 1))
return decimal_value

print(octal_to_decimal('12.5'))  # Output: 10.625
```

- **Hexadecimal to Decimal:**

```
def hex_to_decimal(hex_str):
integer_part, fractional_part = hex_str.split('.')
decimal_value = int(integer_part, 16)
for i, digit in enumerate(fractional_part):
decimal_value += int(digit, 16) * (16 ** -(i + 1))
return decimal_value

print(hex_to_decimal('a.a'))  # Output: 10.625
```

8 Conversions Between Binary, Octal, and Hexadecimal

- **Binary to Octal:**

```
def binary_to_octal(binary_str):
return float_to_octal(binary_to_decimal(binary_str))

print(binary_to_octal('1010.101'))  # Output: '0o12.5'
```

- **Binary to Hexadecimal:**

```
def binary_to_hex(binary_str):
return float_to_hex(binary_to_decimal(binary_str))

print(binary_to_hex('1010.101'))  # Output: '0xa.a'
```

- **Octal to Binary:**

```
def octal_to_binary(octal_str):
return float_to_binary(octal_to_decimal(octal_str))

print(octal_to_binary('12.5'))  # Output: '0b1010.101'
```

- **Hexadecimal to Binary:**

```
def hex_to_binary(hex_str):
return float_to_binary(hex_to_decimal(hex_str))

print(hex_to_binary('a.a'))  # Output: '0b1010.101'
```

Conclusion

Floating-point number conversions require special handling of the fractional part. The methods provided allow accurate conversion between different bases while preserving precision. The int() function helps convert integer parts, while repeated multiplication by the base handles fractional components.

CustomerID	Country	State	City	Zip Code
1	USA	Georgia	Atlanta	30332
2	USA	Georgia	Atlanta	30331
3	USA	Florida	Melbourne	30912
4	USA	Florida	Tampa	30123
5	India	Karnataka	Bangalore	560001
6	India	Maharashtra	Mumbai	578234
7	India	Karnataka	Hubli	569823
8	India	Maharashtra	Mumbai	578234
9	Germany	Bavaria	Munich	80331
10	Canada	Ontario	Toronto	M4B 1B3

Table 2: Customer Information Table

What is a DataFrame?

A **DataFrame** is a two-dimensional, table-like data structure in Pandas, similar to a spreadsheet or SQL table. It consists of rows and columns and allows efficient data manipulation and analysis.

A DataFrame has the following key components:

- **Index**: Unique labels for rows (default is numerical starting from 0).
- **Columns**: Labeled names for different features or attributes.
- **Data**: The actual values stored in the DataFrame.

9 Creating DataFrames in Pandas

Creating a DataFrame from a Dictionary

```
import pandas as pd

data = {
"CustomerID": [1, 2, 3],
"Country": ["USA", "India", "Canada"],
"State": ["Georgia", "Karnataka", "Ontario"],
"City": ["Atlanta", "Bangalore", "Toronto"],
"Zip-Code": ["30332", "560001", "M4B-1B3"]
}

df = pd.DataFrame(data)
print(df)
```

Creating a DataFrame from a List of Lists

```
data = [[1, "USA", "Georgia", "Atlanta", "30332"],
[2, "India", "Karnataka", "Bangalore", "560001"],
[3, "Canada", "Ontario", "Toronto", "M4B-1B3"]]

columns = ["CustomerID", "Country", "State", "City", "Zip-Code"]
df = pd.DataFrame(data, columns=columns)
print(df)
```

Creating a DataFrame from a CSV File

```
df = pd.read_csv("CustomerList.csv")
print(df)
```

Creating a DataFrame from an Excel File

```
df = pd.read_excel("CustomerList.xlsx")
print(df)
```

Creating a DataFrame from a NumPy Array

```
import numpy as np

array = np.array([[1, "USA", "Atlanta"], [2, "India", "Bangalore"], [3, "Canada",
df = pd.DataFrame(array, columns=["CustomerID", "Country", "City"])
print(df)
```

Creating an Empty DataFrame

```
df = pd.DataFrame(columns=["CustomerID", "Country", "State", "City", "Zip-Code"])
print(df)
```

10 Common DataFrame Methods

Displaying Data

```
print(df.head())  # Shows the first 5 rows
print(df.tail(3))  # Shows the last 3 rows
print(df.info())  # Displays column details and data types
print(df.describe())  # Shows statistical summary of numerical columns
```

Accessing Specific Data

```
print(df["Country"])  # Accesses a single column
print(df[["Country", "State"]])  # Accesses multiple columns
print(df.loc[1])  # Accesses a row by index
print(df.iloc[0:2])  # Accesses first two rows using position-based indexing
```

Filtering Data Using Conditions

```
usa_customers = df[df["Country"] == "USA"]  # Filters rows where Country is USA
georgia_customers = df[(df["Country"] == "USA") & (df["State"] == "Georgia")]
print(georgia_customers)
```

Adding New Columns

```
df["Membership"] = ["Gold", "Silver", "Bronze"]  # Adds a new column
print(df)
```

Modifying Values

```
df.at[1, "City"] = "Delhi"  # Updates a specific value
print(df)
```

Deleting Columns and Rows

```
df.drop(columns=["Zip-Code"], inplace=True)  # Deletes a column
df.drop(index=2, inplace=True)  # Deletes a row
print(df)
```

Sorting Data

```
sorted_df = df.sort_values(by="State")  # Sorts by the 'State' column
print(sorted_df)
```

Grouping Data

```
grouped_df = df.groupby("Country").count()  # Groups by country and counts entrie
print(grouped_df)
```

11 Conclusion

DataFrames are a powerful tool for handling structured data in Python. They allow for easy data manipulation, filtering, sorting, and analysis. Mastering Pandas DataFrames will help students efficiently work with large datasets in real-world applications.

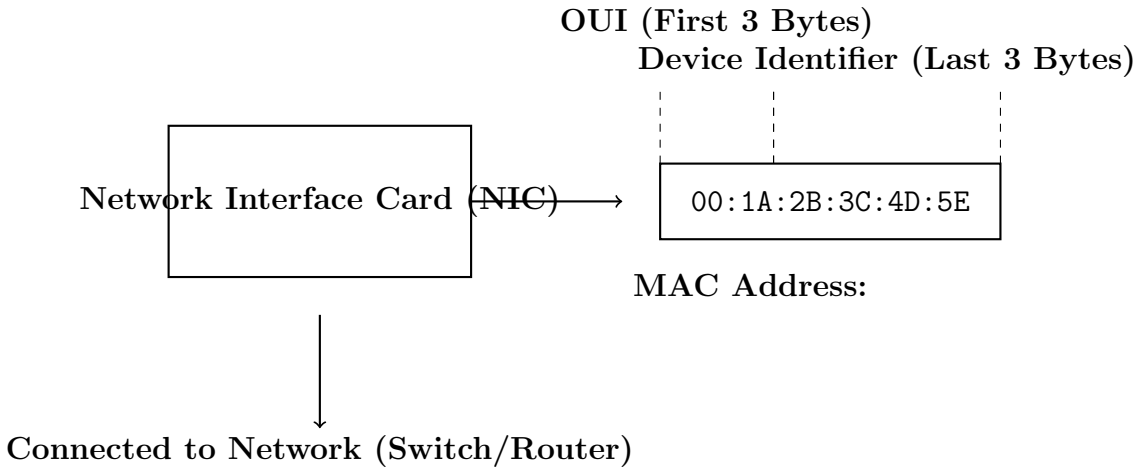
MAC addresses

A **Media Access Control (MAC) address** is a unique identifier assigned to a **network interface card (NIC)** of a device for communication within a network segment. MAC addresses are used in Ethernet, Wi-Fi, and other network technologies to facilitate device identification and communication at the **data link layer (Layer 2)** of the OSI (Open Systems Interconnection Model) model.

Structure of a MAC Address

A MAC address is a **48-bit (6-byte) hexadecimal number**, typically represented in one of the following formats:

- **Colon-separated**: `00:1A:2B:3C:4D:5E`
- **Hyphen-separated**: `00-1A-2B-3C-4D-5E`
- **Dot-separated**: `001A.2B3C.4D5E`




```
import uuid
# Address using uuid and getnode() function
# Making use of hexadecimal number system
print(hex(uuid.getnode()))
#Output
0xf40669da5f06
```