Python Programming Guide - UNIT II: Data Types and Operators

1. Built-in Data Types

Python has several built-in data types that allow you to store different kinds of values. Understanding these is fundamental to programming effectively.

Numeric Types

Integer (int)

Integers are whole numbers without decimal points. They can be positive, negative, or zero.

```
        # Integer examples
        age = 25
        year = 2026
        temperature = -5
        zero = 0
        large_number = 999999999999

        print(type(25))           # Output: <class 'int'>
        print(isinstance(age, int))  # Output: True

        # Integer operations
        a = 10
        b = 3
        print(f"a + b = {a + b}")      # Output: a + b = 13
        print(f"a - b = {a - b}")      # Output: a - b = 7
        print(f"a * b = {a * b}")      # Output: a * b = 30
        print(f"a // b = {a // b}")    # Output: a // b = 3 (integer
division)
        print(f"a % b = {a % b}")      # Output: a % b = 1 (modulo)

        # Binary and hexadecimal representations
        binary_num = 0b1010      # 10 in decimal
        hex_num = 0x1F           # 31 in decimal
        octal_num = 0o17         # 15 in decimal

        print(f"Binary 1010 = {binary_num}")
        print(f"Hex 1F = {hex_num}")
        print(f"Octal 17 = {octal_num}")

        # Large integers (Python handles arbitrary precision)
```

```
huge = 123456789012345678901234567890
print(f"Huge number: {huge}")
print(f"Type: {type(huge)}")  # Still <class 'int'>
```

Real-World Application: Student ID Management

```
# Student ID system
class StudentIDSystem:
    def __init__(self, school_code):
        self.school_code = school_code
        self.next_id = 1000

    def generate_student_id(self, batch_year):
        """Generate unique student ID."""
        student_id = int(f"{batch_year}{self.school_code}
{self.next_id}")
        self.next_id += 1
        return student_id

# Usage
system = StudentIDSystem(school_code=101)
id1 = system.generate_student_id(2026)
id2 = system.generate_student_id(2026)
print(f"Student ID 1: {id1}")
print(f"Student ID 2: {id2}")
```

Float

Floats represent numbers with decimal points. They use the IEEE 754 double-precision format.

```
# Float examples
pi = 3.14159
temperature = 98.6
price = 19.99
scientific = 1.5e-3  # Scientific notation (0.0015)

print(type(3.14))         # Output: <class 'float'>
print(isinstance(pi, float))  # Output: True

# Float arithmetic
x = 10.5
y = 3.2
print(f"x + y = {x + y}")      # Output: x + y = 13.7
print(f"x / y = {x / y:.4f}")  # Output: x / y = 3.2812

# Special float values
infinity = float('inf')
neg_infinity = float('-inf')
not_a_number = float('nan')

print(f"Infinity: {infinity}")
print(f"Negative Infinity: {neg_infinity}")
print(f"NaN: {not_a_number}")
print(f"Is inf? {infinity == float('inf')}")  # Output: True

# Precision issues
result = 0.1 + 0.2
print(result)              # Output: 0.30000000000000004
print(f"{result:.17f}")    # Shows the precision limitation
```

```
        # Rounding
        value = 3.7
        print(f"Round {value}: {round(value)}")  # Output: Round 3.7: 4
        print(f"Round to 2 decimals: {round(3.14159, 2)}")  # Output: 3.14
```

Real-World Application: Financial Calculations

```
        class InvestmentCalculator:
            def __init__(self, principal, rate, time):
                self.principal = principal
                self.rate = rate        # Annual interest rate
                self.time = time        # Years

            def simple_interest(self):
                """Calculate simple interest."""
                interest = (self.principal * self.rate * self.time) / 100
                return interest

            def compound_interest(self, compounds_per_year=12):
                """Calculate compound interest."""
                amount = self.principal * (1 + self.rate / (100 *
compounds_per_year)) ** (compounds_per_year * self.time)
                return amount - self.principal

            def display_report(self):
                """Display investment report."""
                print(f"\nInvestment Report")
                print("=" * 50)
                print(f"Principal: ${self.principal:.2f}")
                print(f"Annual Rate: {self.rate}%")
                print(f"Time Period: {self.time} years")
                print(f"Simple Interest: ${self.simple_interest():.2f}")
                print(f"Compound Interest (monthly):
${self.compound_interest():.2f}")

        # Usage
        investment = InvestmentCalculator(10000, 5.5, 10)
        investment.display_report()
```

Output:

```
Investment Report
==================================================
Principal: $10000.00
Annual Rate: 5.5%
Time Period: 10 years
Simple Interest: $5500.00
Compound Interest (monthly): $7089.49
```

Complex Numbers

Complex numbers have a real and imaginary part (in the form a + bj).

```
        # Complex number examples
        z1 = 3 + 4j          # 3 is real, 4 is imaginary
        z2 = 2 - 1j
        z3 = complex(5, 2)   # Alternative way

        print(type(z1))              # Output: <class 'complex'>
        print(z1)                    # Output: (3+4j)
```

```
print(f"Real part: {z1.real}") # Output: Real part: 3.0
print(f"Imaginary part: {z1.imag}")  # Output: Imaginary part: 4.0

# Complex arithmetic
print(f"z1 + z2 = {z1 + z2}")  # Output: z1 + z2 = (5+3j)
print(f"z1 * z2 = {z1 * z2}")  # Output: z1 * z2 = (10+5j)
print(f"|z1| = {abs(z1)}")     # Output: |z1| = 5.0 (magnitude)

# Complex conjugate
print(f"Conjugate of z1: {z1.conjugate()}")  # Output: (3-4j)
```

Real-World Application: Signal Processing

```
import cmath

class SignalAnalyzer:
    """Analyze signals using complex numbers."""

    @staticmethod
    def impedance(resistance, reactance):
        """Calculate impedance (resistance + reactance)."""
        return complex(resistance, reactance)

    @staticmethod
    def magnitude_phase(impedance):
        """Get magnitude and phase of impedance."""
        magnitude = abs(impedance)
        phase = cmath.phase(impedance)
        return magnitude, phase

# Usage
Z = SignalAnalyzer.impedance(100, 50)  # 100 ohms resistance, 50
ohms reactance
magnitude, phase = SignalAnalyzer.magnitude_phase(Z)
print(f"Impedance: {Z}")
print(f"Magnitude: {magnitude:.2f} ohms")
print(f"Phase: {cmath.degrees(phase):.2f} degrees")
```

Boolean

Booleans represent truth values: True or False. They're used in conditional statements and logical operations.

```
# Boolean examples
is_student = True
is_graduated = False
is_valid = bool(1)           # Any non-zero number is True
is_empty = bool([])          # Empty containers are False
is_nonempty = bool([1, 2, 3])  # Non-empty containers are True

print(type(True))            # Output: <class 'bool'>
print(isinstance(is_student, bool))  # Output: True

# Boolean operations
print(True and False)        # Output: False
print(True or False)         # Output: True
print(not True)              # Output: False

# Boolean expressions
age = 25
```

```
is_adult = age >= 18
print(f"Is adult: {is_adult}")  # Output: Is adult: True

# Truthiness of different types
print(bool(0))              # Output: False
print(bool(1))              # Output: True
print(bool(""))             # Output: False
print(bool("hello"))        # Output: True
print(bool([]))             # Output: False
print(bool([1, 2, 3]))      # Output: True
print(bool(None))           # Output: False
```

Real-World Application: User Authentication

```
class UserAccount:
    def __init__(self, username, password):
        self.username = username
        self.password = password
        self.is_active = True
        self.is_verified = False
        self.login_attempts = 0
        self.max_attempts = 5

    def verify_credentials(self, entered_password):
        """Verify user credentials."""
        if not self.is_active:
            return False

        if self.login_attempts >= self.max_attempts:
            self.is_active = False
            return False

        if entered_password == self.password:
            self.login_attempts = 0
            return True
        else:
            self.login_attempts += 1
            return False

    def display_status(self):
        """Display account status."""
        print(f"\nAccount Status for {self.username}:")
        print(f"  Active: {self.is_active}")
        print(f"  Verified: {self.is_verified}")
        print(f"  Login Attempts:
{self.login_attempts}/{self.max_attempts}")

# Usage
account = UserAccount("alice", "secure_pass123")
print(account.verify_credentials("wrong_pass"))   # False
print(account.verify_credentials("secure_pass123"))  # True
account.display_status()
```

String

Strings are sequences of characters. They’re immutable and can be created with single, double, or triple quotes.

```
# String examples
name = "Alice"
```

```
greeting = 'Hello'
multiline = """This is a
multiline
string"""

print(type("hello"))         # Output: <class 'str'>
print(len("hello"))          # Output: 5 (length)

# String indexing (0-based)
text = "Python"
print(text[0])               # Output: P (first character)
print(text[-1])              # Output: n (last character)
print(text[1:4])             # Output: yth (slicing)

# String methods
sentence = "Hello World"
print(sentence.lower())           # Output: hello world
print(sentence.upper())           # Output: HELLO WORLD
print(sentence.replace("World", "Python"))  # Output: Hello Python
print(sentence.split())           # Output: ['Hello', 'World']

# String concatenation and formatting
first_name = "John"
last_name = "Doe"

# Concatenation
full_name = first_name + " " + last_name
print(full_name)             # Output: John Doe

# F-strings (formatted string literals) - preferred
age = 30
formatted = f"{first_name} is {age} years old"
print(formatted)             # Output: John is 30 years old

# Format method
formatted2 = "{} is {} years old".format(first_name, age)
print(formatted2)            # Output: John is 30 years old

# String with special characters
escaped = "He said \"Hello\""  # Escaped quotes
newline = "Line1\nLine2"
tab = "Column1\tColumn2"
print(escaped)
print(newline)
print(tab)

# Raw strings (ignores escape sequences)
path = r"C:\Users\Alice\Documents"  # Raw string
print(path)                  # Output: C:\Users\Alice\Documents
```

Real-World Application: Text Processing

```
class TextProcessor:
    """Process and analyze text data."""

    @staticmethod
    def analyze_text(text):
        """Analyze text statistics."""
        words = text.split()
        sentences = text.split('.')
```

```
                return {
                    'character_count': len(text),
                    'word_count': len(words),
                    'sentence_count': len([s for s in sentences if
s.strip()]),
                    'average_word_length': len(text) / len(words) if words
else 0
                }

            @staticmethod
            def format_report(text):
                """Format text as a report."""
                analysis = TextProcessor.analyze_text(text)
                report = f"""
        TEXT ANALYSIS REPORT
        {'=' * 40}
        Characters: {analysis['character_count']}
        Words: {analysis['word_count']}
        Sentences: {analysis['sentence_count']}
        Avg Word Length: {analysis['average_word_length']:.2f}
                """
                return report

        # Usage
        sample_text = "Python is a powerful language. It is easy to learn.
Python is widely used."
        print(TextProcessor.format_report(sample_text))
```

---

2. Type Conversion

Type conversion (casting) is the process of converting one data type to another.

Implicit Type Conversion

Python automatically converts compatible types in certain situations.

```
        # Integer to float conversion
        x = 10
        y = 5.5
        result = x + y              # x is converted to float
        print(result)               # Output: 15.5
        print(type(result))         # Output: <class 'float'>

        # String and integer concatenation (automatic in some operations)
        # Note: This causes error, so we need explicit conversion
        # print("Age: " + 25)  # TypeError

        # Boolean to integer
        print(True + 5)             # Output: 6 (True is 1)
        print(False + 5)            # Output: 5 (False is 0)
        print(True + True)          # Output: 2

        # Implicit conversion in comparisons
        print(5 == 5.0)             # Output: True
        print(1 == True)            # Output: True
        print(0 == False)           # Output: True
```

Explicit Type Conversion

You explicitly convert types using conversion functions.

```
        # int() - Convert to integer
        print(int(5.7))             # Output: 5 (truncates decimal)
        print(int("123"))           # Output: 123
        print(int("FF", 16))        # Output: 255 (hexadecimal)
        print(int(True))            # Output: 1

        # float() - Convert to float
        print(float(5))             # Output: 5.0
        print(float("3.14"))        # Output: 3.14
        print(float("inf"))         # Output: inf

        # str() - Convert to string
        print(str(123))             # Output: '123'
        print(str(3.14))            # Output: '3.14'
        print(str(True))            # Output: 'True'
        print(str([1, 2, 3]))       # Output: '[1, 2, 3]'

        # bool() - Convert to boolean
        print(bool(1))              # Output: True
        print(bool(0))              # Output: False
        print(bool(""))             # Output: False
        print(bool("text"))         # Output: True

        # complex() - Convert to complex
        print(complex(5))           # Output: (5+0j)
        print(complex(3, 4))        # Output: (3+4j)
```

Real-World Application: User Input Validation

```
        class FormValidator:
            """Validate and convert form inputs."""

            @staticmethod
            def get_integer(prompt, min_val=None, max_val=None):
                """Get and validate integer input."""
                while True:
                    try:
                        value = int(input(prompt))
                        if min_val is not None and value < min_val:
                            print(f"Value must be at least {min_val}")
                            continue
                        if max_val is not None and value > max_val:
                            print(f"Value must be at most {max_val}")
                            continue
                        return value
                    except ValueError:
                        print("Invalid input. Please enter a valid
integer.")

            @staticmethod
            def get_float(prompt, min_val=None, max_val=None):
                """Get and validate float input."""
                while True:
                    try:
                        value = float(input(prompt))
                        if min_val is not None and value < min_val:
                            print(f"Value must be at least {min_val}")
```

```
                        continue
                    if max_val is not None and value > max_val:
                        print(f"Value must be at most {max_val}")
                        continue
                    return value
                except ValueError:
                    print("Invalid input. Please enter a valid number.")

        @staticmethod
        def get_choice(prompt, valid_options):
            """Get validated choice from user."""
            while True:
                choice = input(prompt).strip().upper()
                if choice in valid_options:
                    return choice
                print(f"Invalid choice. Choose from: {',
'.join(valid_options)}")

    # Usage example (simulated)
    print("Example usage: FormValidator.get_integer('Enter age (0-100):
', 0, 100)")
```

---

3. Input and Output Functions

Output Function: print()

The `print()` function sends output to the console.

```
    # Basic print
    print("Hello, World!")           # Output: Hello, World!

    # Multiple arguments
    print("Name:", "Alice", "Age:", 25)  # Output: Name: Alice Age: 25

    # Separator and end parameters
    print("A", "B", "C", sep="-")        # Output: A-B-C
    print("Line1")
    print("Line2", end=" ")             # No newline after
    print("Line2_continued")            # Output: Line1 / Line2
Line2_continued

    # Formatted output
    pi = 3.14159
    print(f"Pi value: {pi}")            # Output: Pi value: 3.14159
    print(f"Pi to 2 decimals: {pi:.2f}")  # Output: Pi to 2 decimals:
3.14

    # Using format()
    print("Value: {}".format(100))      # Output: Value: 100
    print("{0} + {1} = {2}".format(2, 3, 5))  # Output: 2 + 3 = 5

    # String repetition in print
    print("*" * 50)                     # Output: 50 asterisks
    print("Hello\n" * 3)               # Output: Hello printed 3 times
with newlines
```

Real-World Application: Pretty Report Generation

```
        class ReportGenerator:
            """Generate formatted reports."""

            @staticmethod
            def print_header(title, width=60):
                """Print a formatted header."""
                print("=" * width)
                print(title.center(width))
                print("=" * width)

            @staticmethod
            def print_table(headers, rows):
                """Print formatted table."""
                col_widths = [max(len(str(h)), max(len(str(r[i])) for r in
rows))
                              for i, h in enumerate(headers)]

                # Header
                header_row = " | ".join(h.ljust(w) for h, w in zip(headers,
col_widths))
                print(header_row)
                print("-" * len(header_row))

                # Data rows
                for row in rows:
                    data_row = " | ".join(str(d).ljust(w) for d, w in
zip(row, col_widths))
                    print(data_row)

            @staticmethod
            def print_sales_report():
                """Print a sample sales report."""
                ReportGenerator.print_header("QUARTERLY SALES REPORT - Q1
2026")

                headers = ["Product", "Units", "Price", "Total"]
                rows = [
                    ["Laptop", 15, "$1200", "$18000"],
                    ["Mouse", 50, "$25", "$1250"],
                    ["Keyboard", 30, "$75", "$2250"],
                    ["Monitor", 20, "$400", "$8000"]
                ]

                ReportGenerator.print_table(headers, rows)
                print(f"\nTotal Revenue: ${(18000 + 1250 + 2250 + 8000):,}")

        # Usage
        ReportGenerator.print_sales_report()
```

Input Function: input()

The `input()` function reads a line of text from the user.

```
        # Basic input
        name = input("Enter your name: ")
        print(f"Hello, {name}!")

        # Input always returns a string
        age_str = input("Enter your age: ")
        print(f"Type of input: {type(age_str)}")  # <class 'str'>
```

```
        # Convert input to other types
        age = int(input("Enter your age: "))
        height = float(input("Enter your height (in meters): "))
        is_student = input("Are you a student? (yes/no): ").lower() == "yes"

        # Multiple inputs
        data = input("Enter name, age, city (separated by commas):
").split(",")
        name, age, city = data
        print(f"{name.strip()} is {age.strip()} years old and lives in
{city.strip()}")
```

Real-World Application: Interactive User Form

```
        class UserForm:
            """Interactive form for user data collection."""

            @staticmethod
            def collect_user_info():
                """Collect user information interactively."""
                print("\n" + "=" * 50)
                print("USER REGISTRATION FORM")
                print("=" * 50)

                name = input("Full Name: ").strip()

                while True:
                    try:
                        age = int(input("Age: "))
                        if 0 < age < 150:
                            break
                        else:
                            print("Please enter a valid age (1-149)")
                    except ValueError:
                        print("Age must be a number")

                email = input("Email: ").strip()

                while True:
                    gender = input("Gender (M/F/Other): ").strip().upper()
                    if gender in ["M", "F", "OTHER"]:
                        break
                    print("Please enter M, F, or Other")

                country = input("Country: ").strip()

                return {
                    'name': name,
                    'age': age,
                    'email': email,
                    'gender': gender,
                    'country': country
                }

            @staticmethod
            def display_summary(user_data):
                """Display collected information."""
                print("\n" + "=" * 50)
                print("REGISTRATION SUMMARY")
                print("=" * 50)
```

```
            for key, value in user_data.items():
                print(f"{key.capitalize():15}: {value}")

    # Usage (simulated - requires actual user input)
    # user_info = UserForm.collect_user_info()
    # UserForm.display_summary(user_info)
```

---

4. Operators in Python

Arithmetic Operators

Arithmetic operators perform mathematical operations.

```
    a = 10
    b = 3

    # Addition
    print(f"{a} + {b} = {a + b}")           # Output: 10 + 3 = 13

    # Subtraction
    print(f"{a} - {b} = {a - b}")           # Output: 10 - 3 = 7

    # Multiplication
    print(f"{a} * {b} = {a * b}")           # Output: 10 * 3 = 30

    # Division (returns float)
    print(f"{a} / {b} = {a / b}")           # Output: 10 / 3 = 3.3333...

    # Floor Division (rounds down)
    print(f"{a} // {b} = {a // b}")         # Output: 10 // 3 = 3

    # Modulo (remainder)
    print(f"{a} % {b} = {a % b}")           # Output: 10 % 3 = 1

    # Exponentiation (power)
    print(f"{a} ** {b} = {a ** b}")         # Output: 10 ** 3 = 1000

    # String and list multiplication
    print("Ha" * 3)                         # Output: HaHaHa
    print([1, 2] * 3)                       # Output: [1, 2, 1, 2, 1, 2]

    # Arithmetic with mixed types
    x = 10
    y = 3.0
    print(f"{x} + {y} = {x + y}")           # Output: 10 + 3.0 = 13.0
(result is float)
```

Real-World Application: Engineering Calculations

```
    import math

    class StructuralCalculator:
        """Calculate structural properties."""

        @staticmethod
        def beam_bending_moment(force, distance):
            """Calculate bending moment."""
            return force * distance
```

```
            @staticmethod
            def beam_shear_stress(force, area):
                """Calculate shear stress."""
                return force / area

            @staticmethod
            def circle_properties(radius):
                """Calculate circle properties."""
                return {
                    'area': math.pi * radius ** 2,
                    'circumference': 2 * math.pi * radius,
                    'diameter': 2 * radius
                }

            @staticmethod
            def triangle_area(base, height):
                """Calculate triangle area."""
                return 0.5 * base * height

        # Usage
        print("Bending Moment (100N at 5m):",
StructuralCalculator.beam_bending_moment(100, 5), "N·m")
        print("Shear Stress (500N on 100m²):",
StructuralCalculator.beam_shear_stress(500, 100), "Pa")
        circle = StructuralCalculator.circle_properties(5)
        print(f"Circle (r=5): Area={circle['area']:.2f}, Circumference=
{circle['circumference']:.2f}")
```

Relational (Comparison) Operators

Comparison operators return Boolean values.

```
        a = 10
        b = 5

        # Equal
        print(f"a == b: {a == b}")              # Output: False

        # Not equal
        print(f"a != b: {a != b}")              # Output: True

        # Greater than
        print(f"a > b: {a > b}")                # Output: True

        # Less than
        print(f"a < b: {a < b}")                # Output: False

        # Greater than or equal
        print(f"a >= b: {a >= b}")              # Output: True

        # Less than or equal
        print(f"a <= b: {a <= b}")              # Output: False

        # String comparison
        print(f"'abc' < 'def': {'abc' < 'def'}")  # Output: True
(alphabetical order)
        print(f"'hello' == 'hello': {'hello' == 'hello'}")  # Output: True

        # Comparison chaining
```

```
        x = 5
        print(0 < x < 10)                       # Output: True
        print(x == 5 and x < 10)                # Same as above, but less
readable

        # List comparison
        list1 = [1, 2, 3]
        list2 = [1, 2, 3]
        list3 = list1
        print(list1 == list2)                   # Output: True (same
content)
        print(list1 is list2)                   # Output: False (different
objects)
        print(list1 is list3)                   # Output: True (same object)
```

Real-World Application: Grade Evaluation

```
        class GradeEvaluator:
            """Evaluate student grades."""

            # Grade boundaries
            EXCELLENT = 90
            GOOD = 80
            SATISFACTORY = 70
            PASS = 60

            @staticmethod
            def get_grade_letter(score):
                """Convert numerical score to letter grade."""
                if score >= GradeEvaluator.EXCELLENT:
                    return 'A'
                elif score >= GradeEvaluator.GOOD:
                    return 'B'
                elif score >= GradeEvaluator.SATISFACTORY:
                    return 'C'
                elif score >= GradeEvaluator.PASS:
                    return 'D'
                else:
                    return 'F'

            @staticmethod
            def evaluate_student(score):
                """Provide evaluation feedback."""
                if score >= GradeEvaluator.EXCELLENT:
                    return f"Score {score}: Excellent! Outstanding
performance."
                elif score >= GradeEvaluator.GOOD:
                    return f"Score {score}: Good! Keep up the great work."
                elif score >= GradeEvaluator.SATISFACTORY:
                    return f"Score {score}: Satisfactory. Room for
improvement."
                elif score >= GradeEvaluator.PASS:
                    return f"Score {score}: Passed, but needs more effort."
                else:
                    return f"Score {score}: Failed. Please retake the exam."

        # Usage
        scores = [95, 85, 75, 65, 55]
        for score in scores:
            letter = GradeEvaluator.get_grade_letter(score)
            evaluation = GradeEvaluator.evaluate_student(score)
```

```
            print(f"Grade {letter}: {evaluation}")
```

Logical Operators

Logical operators combine Boolean values.

```
        # AND operator
        print(True and True)                    # Output: True
        print(True and False)                   # Output: False
        print(False and False)                  # Output: False

        # OR operator
        print(True or False)                    # Output: True
        print(False or False)                   # Output: False

        # NOT operator
        print(not True)                         # Output: False
        print(not False)                        # Output: True

        # Combining conditions
        age = 25
        income = 50000

        is_valid_applicant = (age >= 18) and (income >= 30000)
        print(f"Valid applicant: {is_valid_applicant}")  # Output: True

        has_discount = (age < 13) or (age > 60) or (income < 20000)
        print(f"Eligible for discount: {has_discount}")  # Output: False

        is_risky = (age < 25) and (income < 30000)
        print(f"High risk applicant: {is_risky}")  # Output: False

        # Short-circuit evaluation
        def check_admin(user_dict):
            """Check if user is admin."""
            return user_dict and user_dict.get('role') == 'admin'

        print(check_admin(None))                # Output: False (short-
circuit)
        print(check_admin({'role': 'admin'}))   # Output: True
        print(check_admin({'role': 'user'}))    # Output: False
```

Real-World Application: Loan Eligibility Checker

```
        class LoanEligibility:
            """Check loan eligibility based on criteria."""

            MIN_AGE = 21
            MIN_INCOME = 30000
            MIN_CREDIT_SCORE = 650
            MAX_DEBT_RATIO = 0.4  # 40% of income

            @staticmethod
            def check_eligibility(applicant):
                """Check if applicant qualifies for loan."""
                age = applicant.get('age', 0)
                income = applicant.get('income', 0)
                credit_score = applicant.get('credit_score', 0)
                existing_debt = applicant.get('existing_debt', 0)
```

```
                # All conditions must be met
                age_check = age >= LoanEligibility.MIN_AGE
                income_check = income >= LoanEligibility.MIN_INCOME
                credit_check = credit_score >=
LoanEligibility.MIN_CREDIT_SCORE
                debt_ratio = existing_debt / income if income > 0 else 1
                debt_check = debt_ratio <= LoanEligibility.MAX_DEBT_RATIO

                eligible = age_check and income_check and credit_check and
debt_check

                return {
                    'eligible': eligible,
                    'age_check': age_check,
                    'income_check': income_check,
                    'credit_check': credit_check,
                    'debt_check': debt_check
                }

            @staticmethod
            def display_result(applicant_name, result):
                """Display eligibility result."""
                print(f"\nLoan Eligibility Report for {applicant_name}")
                print("-" * 40)
                print(f"Age Check (>={LoanEligibility.MIN_AGE}):
{result['age_check']}")
                print(f"Income Check (>=${LoanEligibility.MIN_INCOME}):
{result['income_check']}")
                print(f"Credit Score Check (>=
{LoanEligibility.MIN_CREDIT_SCORE}): {result['credit_check']}")
                print(f"Debt Ratio Check
(≤{LoanEligibility.MAX_DEBT_RATIO*100}%): {result['debt_check']}")
                print("-" * 40)
                status = "APPROVED" if result['eligible'] else "REJECTED"
                print(f"Status: {status}")

        # Usage
        applicants = [
            {'name': 'Alice', 'age': 28, 'income': 50000, 'credit_score':
750, 'existing_debt': 10000},
            {'name': 'Bob', 'age': 45, 'income': 35000, 'credit_score': 600,
'existing_debt': 20000},
        ]

        for app in applicants:
            result = LoanEligibility.check_eligibility(app)
            LoanEligibility.display_result(app['name'], result)
```

Assignment Operators

Assignment operators assign values to variables.

```
        # Simple assignment
        x = 10
        print(x)                                # Output: 10

        # Add and assign
        x += 5      # x = x + 5
        print(x)                                # Output: 15
```

```
# Subtract and assign
x -= 3      # x = x - 3
print(x)                                # Output: 12

# Multiply and assign
x *= 2      # x = x * 2
print(x)                                # Output: 24

# Divide and assign
x /= 4      # x = x / 4
print(x)                                # Output: 6.0

# Floor divide and assign
x //= 2     # x = x // 2
print(x)                                # Output: 3.0

# Modulo and assign
x %= 2      # x = x % 2
print(x)                                # Output: 1.0

# Exponent and assign
x **= 3     # x = x ** 3
print(x)                                # Output: 1.0

# Multiple assignment
a = b = c = 10
print(a, b, c)                          # Output: 10 10 10

# Unpacking assignment
x, y, z = 1, 2, 3
print(x, y, z)                          # Output: 1 2 3

x, y = y, x                             # Swap
print(x, y)                             # Output: 2 1
```

Membership Operators

Membership operators check if a value exists in a sequence.

```
# in operator
fruits = ['apple', 'banana', 'cherry']
print('apple' in fruits)                # Output: True
print('grape' in fruits)                # Output: False

# not in operator
print('grape' not in fruits)            # Output: True
print('apple' not in fruits)            # Output: False

# With strings
text = "Hello World"
print('H' in text)                      # Output: True
print('xyz' in text)                    # Output: False

# With dictionaries
person = {'name': 'Alice', 'age': 25}
print('name' in person)                 # Output: True
print('name' in person.values())        # Output: False (checking
values)
print('Alice' in person.values())       # Output: True
```

```
        # With ranges
        print(5 in range(1, 10))                # Output: True
        print(15 in range(1, 10))               # Output: False
```

Real-World Application: Inventory Management

```
        class InventoryManager:
            """Manage product inventory."""

            def __init__(self):
                self.inventory = {
                    'laptop': {'quantity': 5, 'price': 1000},
                    'mouse': {'quantity': 50, 'price': 25},
                    'keyboard': {'quantity': 30, 'price': 75}
                }

            def product_exists(self, product_name):
                """Check if product is in inventory."""
                return product_name.lower() in self.inventory

            def is_in_stock(self, product_name, required_quantity=1):
                """Check if product has sufficient stock."""
                if self.product_exists(product_name):
                    current = self.inventory[product_name.lower()]
['quantity']
                    return current >= required_quantity
                return False

            def check_products(self, products_to_check):
                """Check multiple products."""
                print("\nInventory Status:")
                print("-" * 40)
                for product in products_to_check:
                    exists = self.product_exists(product)
                    status = "Available" if exists else "Not Available"
                    print(f"{product}: {status}")

        # Usage
        inventory = InventoryManager()
        inventory.check_products(['laptop', 'phone', 'mouse', 'tablet'])
        print(f"\nLaptop in stock: {inventory.is_in_stock('laptop')}")
        print(f"10 keyboards available: {inventory.is_in_stock('keyboard',
10)}")
```

Identity Operators

Identity operators compare memory addresses (object identity).

```
        # is operator (same object)
        a = [1, 2, 3]
        b = [1, 2, 3]
        c = a

        print(a == b)                           # Output: True (same
content)
        print(a is b)                           # Output: False (different
objects)
        print(a is c)                           # Output: True (same object)

        # With integers (optimization in Python)
```

```
        x = 5
        y = 5
        print(x is y)                           # Output: True (small
integers cached)

        x = 256
        y = 256
        print(x is y)                           # Output: True (caching up
to 256)

        x = 257
        y = 257
        print(x is y)                           # Output: False (no caching
beyond 256)

        # With None
        value = None
        print(value is None)                    # Output: True
        print(value is not None)                # Output: False

        # With booleans
        flag = True
        print(flag is True)                     # Output: True

        # Practical use
        def process_data(data=None):
            """Process data if provided."""
            if data is None:
                print("No data provided, using default")
                data = []
            return data

        result1 = process_data()
        result2 = process_data([1, 2, 3])
```

---

5. Operator Precedence and Associativity

Operator precedence determines the order in which operations are evaluated. Operators with higher precedence are evaluated first.

Precedence Order (Highest to Lowest)

```
        # 1. Parentheses
        result = (2 + 3) * 4          # Output: 20 (not 14)

        # 2. Exponentiation
        result = 2 ** 3 ** 2          # Output: 512 (evaluated right to
left: 2^(3^2) = 2^9)

        # 3. Unary +, -, ~
        result = -2 ** 2              # Output: -4 (evaluated as -(2^2))
        result = (-2) ** 2            # Output: 4

        # 4. *, /, //, %
        result = 10 + 2 * 3           # Output: 16 (multiplication before
addition)
        result = 10 / 2 * 3           # Output: 15 (left to right)
```

```
# 5. +, - (binary)
result = 10 + 5 - 3           # Output: 12 (left to right)

# 6. <<, >> (bitwise shifts)
result = 5 << 1               # Output: 10 (bitwise left shift)

# 7. &, ^, |, (bitwise)
result = 5 & 3                # Output: 1 (bitwise AND)

# 8. ==, !=, <, >, <=, >= (comparisons)
result = 5 > 3 and 10 < 20    # Output: True

# 9. not
result = not (5 > 3)          # Output: False

# 10. and
result = True and False or True  # Output: True (and before or)

# 11. or
result = False or True        # Output: True

# 12. =, +=, -= etc. (assignment)
x = 5
```

Associativity

Most operators are left-associative (evaluated left to right), except exponentiation which is right-associative.

```
# Left-associative
result = 10 - 5 - 2           # (10 - 5) - 2 = 3 (not 10 - (5 - 2) = 7)
result = 20 / 4 / 2           # (20 / 4) / 2 = 2.5

# Right-associative (exponentiation)
result = 2 ** 3 ** 2          # 2 ** (3 ** 2) = 2 ** 9 = 512

# Left-associative (comparison)
result = 1 < 2 < 3            # (1 < 2) and (2 < 3) = True
```

Real-World Application: Complex Formula Evaluation

```
class MathematicalCalculator:
    """Evaluate complex mathematical expressions."""

    @staticmethod
    def calculate_quadratic(a, b, c, x):
        """Calculate ax² + bx + c"""
        # Proper precedence: exponentiation before multiplication
        return a * x**2 + b * x + c

    @staticmethod
    def calculate_pythagorean(a, b):
        """Calculate c = √(a² + b²)"""
        # Proper precedence: exponentiation before addition
        import math
        return math.sqrt(a**2 + b**2)

    @staticmethod
    def compound_interest(principal, rate, time, compounds):
```

```
                """Calculate A = P(1 + r/n)^(nt)"""
                # Proper operator precedence crucial here
                return principal * (1 + rate / (100 * compounds)) **
(compounds * time)

            @staticmethod
            def demonstrate_precedence():
                """Show importance of operator precedence."""
                print("Operator Precedence Examples:")
                print("=" * 50)

                # Example 1
                x = 5
                result1 = 2 * x ** 2 + 3 * x + 1
                print(f"2*x² + 3*x + 1 where x={x}: {result1}")
                # Breakdown: 2*25 + 15 + 1 = 50 + 15 + 1 = 66

                # Example 2
                a, b, c = 1, -5, 6
                x = 3
                result2 = a * x**2 + b * x + c
                print(f"Quadratic with a={a}, b={b}, c={c}, x={x}:
{result2}")

                # Example 3
                principal, rate, time, compounds = 10000, 5, 10, 12
                result3 =
MathematicalCalculator.compound_interest(principal, rate, time,
compounds)
                print(f"Compound Interest (P={principal}, r={rate}%, t=
{time}y, n={compounds}): ${result3:.2f}")

        # Usage
        MathematicalCalculator.demonstrate_precedence()
```

Summary of Unit II

This unit covers all the fundamental data types and operators:

- **Data Types**: int, float, complex, bool, str
- **Type Conversion**: Implicit and explicit conversion between types
- **Input/Output**: Using `input()` and `print()` functions
- **Operators**: Arithmetic, relational, logical, assignment, membership, and identity
- **Precedence and Associativity**: Proper order of operations

---

Practice Exercises for Unit II

1. Create a type conversion calculator that accepts user input and converts between different types
2. Write a program that uses all comparison operators to compare different data types
3. Build an expression evaluator that demonstrates operator precedence
4. Create a form validator using type conversion and logical operators
5. Write a program using membership operators to search in different

data structures

Next Unit: Unit III - Control Flow and Functions