Tutorial by Dr. Ayes Chinmay
Date: 21/10/2025

Python Chapter 10: Decimal, Money, and Other Classes - Comprehensive Tutorial

This tutorial provides a detailed exploration of Chapter 10, covering all sections from 10.1 to 10.14 with comprehensive code examples, explanations, and outputs.

Table of Contents

1. 10.1 Overview of Numeric Classes
2. 10.2 Limitations of Floating-Point Format
3. 10.3 Introducing the Decimal Class
4. 10.4 Special Operations on Decimal Objects
5. 10.5 A Decimal Class Application
6. 10.6 Designing a Money Class
7. 10.7 Writing the Basic Money Class (Containment)
8. 10.8 Displaying Money Objects (**str**, **repr**)
9. 10.9 Other Monetary Operations
10. 10.10 Demo: A Money Calculator
11. 10.11 Setting the Default Currency
12. 10.12 Money and Inheritance
13. 10.13 The Fraction Class
14. 10.14 The Complex Class

10.1 Overview of Numeric Classes

Python provides several built-in numeric types, each designed for specific use cases. Understanding when and how to use these different numeric classes is crucial for writing efficient and accurate programs.

Built-in Numeric Types

Python's primary numeric types include:

- **int**: Integer values with unlimited precision
- **float**: Floating-point numbers (IEEE 754 double precision)
- **complex**: Complex numbers with real and imaginary parts
- **bool**: Boolean values (subclass of int)

Example: Basic Numeric Types

```
# Integer
age = 25
print(f"Age: {age}, Type: {type(age)}")

# Float
temperature = 98.6
print(f"Temperature: {temperature}, Type: {type(temperature)}")

# Complex
impedance = 3 + 4j
print(f"Impedance: {impedance}, Type: {type(impedance)}")

# Boolean
is_valid = True
print(f"Is Valid: {is_valid}, Type: {type(is_valid)}")
print(f"Boolean as int: {int(is_valid)}")
```

Output:

```
Age: 25, Type: &lt;class 'int'&gt;
Temperature: 98.6, Type: &lt;class 'float'&gt;
Impedance: (3+4j), Type: &lt;class 'complex'&gt;
Is Valid: True, Type: &lt;class 'bool'&gt;
Boolean as int: 1
```

Extended Numeric Types from Modules

Python also provides specialized numeric types through modules:

- **decimal.Decimal**: Exact decimal arithmetic
- **fractions.Fraction**: Rational number arithmetic
- **Custom Money classes**: For financial calculations

10.2 Limitations of Floating-Point Format

Floating-point numbers, while convenient for most calculations, have inherent limitations that can cause precision issues in critical applications, especially financial software.

The Precision Problem

```
# Demonstrating floating-point precision issues
print("Floating-point precision problems:")
print(f"0.1 + 0.2 = {0.1 + 0.2}")
print(f"0.1 + 0.2 == 0.3: {0.1 + 0.2 == 0.3}")

# More examples
result1 = 0.1 + 0.1 + 0.1
```

```
print(f"0.1 + 0.1 + 0.1 = {result1}")
print(f"Result equals 0.3: {result1 == 0.3}")

# Financial calculation example
price1 = 19.95
price2 = 0.05
total = price1 + price2
print(f"${price1} + ${price2} = ${total}")
print(f"Is total exactly $20.00? {total == 20.00}")
```

Output:

```
Floating-point precision problems:
0.1 + 0.2 = 0.30000000000000004
0.1 + 0.2 == 0.3: False
0.1 + 0.1 + 0.1 = 0.30000000000000004
Result equals 0.3: False
$19.95 + $0.05 = $20.0
Is total exactly $20.00? True
```

Why This Happens

```
# Understanding binary representation limitations
import sys

print("Understanding floating-point representation:")
print(f"Float info: {sys.float_info}")
print(f"0.1 in binary representation:")

# Convert to binary fraction representation
def float_to_binary_fraction(f):
    """Convert float to binary fraction representation"""
    if f &gt;= 1:
        return bin(int(f))[2:] + '.' + bin(int((f % 1) * (2**52)))[2:]
    else:
        binary = ""
        while f != 0 and len(binary) &lt; 60:
            f *= 2
            if f &gt;= 1:
                binary += '1'
                f -= 1
            else:
                binary += '0'
        return '0.' + binary

print(f"0.1 exact representation: {0.1:.55f}")
print(f"0.2 exact representation: {0.2:.55f}")
print(f"0.3 exact representation: {0.3:.55f}")
```

Output:

```
Understanding floating-point representation:
Float info: sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308, mir
0.1 exact representation: 0.1000000000000000055511151231257827021181583404541015625
0.2 exact representation: 0.2000000000000000111022302462515654042363166809082031250
0.3 exact representation: 0.2999999999999999888977697537484345957636833190917968750
```

10.3 Introducing the Decimal Class

The Decimal class provides exact decimal arithmetic, solving many precision issues inherent with floating-point numbers.

Basic Decimal Usage

```
from decimal import Decimal, getcontext

print("=== Basic Decimal Operations ===")

# Creating Decimal objects
d1 = Decimal('0.1')
d2 = Decimal('0.2')
d3 = Decimal('0.3')

print(f"Decimal('0.1'): {d1}")
print(f"Decimal('0.2'): {d2}")
print(f"d1 + d2 = {d1 + d2}")
print(f"d1 + d2 == Decimal('0.3'): {d1 + d2 == d3}")

# Comparison with float
print(f"\nComparison with float:")
print(f"Float: 0.1 + 0.2 = {0.1 + 0.2}")
print(f"Decimal: 0.1 + 0.2 = {d1 + d2}")
```

Output:

```
=== Basic Decimal Operations ===
Decimal('0.1'): 0.1
Decimal('0.2'): 0.2
d1 + d2 = 0.3
d1 + d2 == Decimal('0.3'): True

Comparison with float:
Float: 0.1 + 0.2 = 0.30000000000000004
Decimal: 0.1 + 0.2 = 0.3
```

Decimal Context and Precision

```
# Working with context and precision
print("=== Decimal Context ===")
print(f"Current context: {getcontext()}")

# Setting precision
getcontext().prec = 4
print(f"\nAfter setting precision to 4:")
print(f"Context: {getcontext()}")

# Calculations with limited precision
d1 = Decimal('1') / Decimal('3')
print(f"1/3 with precision 4: {d1}")

# Reset to higher precision
getcontext().prec = 28
d2 = Decimal('1') / Decimal('3')
print(f"1/3 with precision 28: {d2}")

# Using localcontext for temporary precision changes
from decimal import localcontext

print(f"\n=== Using localcontext ===")
with localcontext() as ctx:
    ctx.prec = 10
    result = Decimal('1') / Decimal('7')
    print(f"1/7 with precision 10: {result}")

# Outside context, precision reverts
result2 = Decimal('1') / Decimal('7')
print(f"1/7 with default precision: {result2}")
```

Output:

```
=== Decimal Context ===
Current context: Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999, ca

After setting precision to 4:
Context: Context(prec=4, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999, capitals=1,
1/3 with precision 4: 0.3333
1/3 with precision 28: 0.3333333333333333333333333333

=== Using localcontext ===
1/7 with precision 10: 0.1428571429
1/7 with default precision: 0.1428571428571428571428571429
```

Creating Decimals from Different Sources

```
print("=== Creating Decimals ===")

# From string (recommended)
d1 = Decimal('3.14159')
print(f"From string '3.14159': {d1}")

# From integer
d2 = Decimal(42)
print(f"From integer 42: {d2}")

# From float (not recommended due to precision issues)
d3 = Decimal(3.14159)
print(f"From float 3.14159: {d3}")

# Better approach: convert float to string first
d4 = Decimal(str(3.14159))
print(f"From str(float): {d4}")

# From tuple (sign, digits, exponent)
d5 = Decimal((0, (3, 1, 4, 1, 5, 9), -5))  # 3.14159
print(f"From tuple: {d5}")

# Special values
inf = Decimal('Infinity')
neg_inf = Decimal('-Infinity')
nan = Decimal('NaN')

print(f"Infinity: {inf}")
print(f"Negative Infinity: {neg_inf}")
print(f"Not a Number: {nan}")
```

Output:

```
=== Creating Decimals ===
From string '3.14159': 3.14159
From integer 42: 42
From float 3.14159: 3.1415899999999999305755649340525269508361816406250
From str(float): 3.14159
From tuple: 3.14159
Infinity: Infinity
Negative Infinity: -Infinity
Not a Number: NaN
```

10.4 Special Operations on Decimal Objects

Decimal objects support many special operations beyond basic arithmetic, providing fine-grained control over numeric calculations.

Quantize and Normalize Operations

```
from decimal import Decimal, ROUND_UP, ROUND_DOWN, ROUND_HALF_UP

print("=== Quantize Operations ===")

# Quantize to specific decimal places
price = Decimal('19.999')
print(f"Original price: {price}")
print(f"Quantized to 2 decimal places: {price.quantize(Decimal('0.01'))}")
print(f"Quantized with ROUND_UP: {price.quantize(Decimal('0.01'), rounding=ROUND_UP)}")
print(f"Quantized with ROUND_DOWN: {price.quantize(Decimal('0.01'), rounding=ROUND_DOWN)}

# Working with currency
tax_rate = Decimal('0.08375')  # 8.375% tax
subtotal = Decimal('127.50')
tax = (subtotal * tax_rate).quantize(Decimal('0.01'))
total = subtotal + tax

print(f"\n=== Currency Example ===")
print(f"Subtotal: ${subtotal}")
print(f"Tax (8.375%): ${tax}")
print(f"Total: ${total}")

# Normalize removes trailing zeros
print(f"\n=== Normalize ===")
d1 = Decimal('1.200')
d2 = Decimal('1.000')
print(f"Before normalize: {d1}, {d2}")
print(f"After normalize: {d1.normalize()}, {d2.normalize()}")
```

Output:

```
=== Quantize Operations ===
Original price: 19.999
Quantized to 2 decimal places: 20.00
Quantized with ROUND_UP: 20.00
Quantized with ROUND_DOWN: 19.99

=== Currency Example ===
Subtotal: $127.50
Tax (8.375%): $10.68
Total: $138.18

=== Normalize ===
Before normalize: 1.200, 1.000
After normalize: 1.2, 1
```

Comparison and Logical Operations

```
print("=== Decimal Comparison Methods ===")

d1 = Decimal('10.5')
d2 = Decimal('10.50')
d3 = Decimal('10.500')

# Standard comparison
print(f"d1 == d2: {d1 == d2}")
print(f"d1 is d2: {d1 is d2}")

# Compare method
print(f"d1.compare(d2): {d1.compare(d2)}")  # 0 means equal
print(f"d1.compare(Decimal('5')): {d1.compare(Decimal('5'))}")  # 1 means greater
print(f"d1.compare(Decimal('15')): {d1.compare(Decimal('15'))}")  # -1 means less

# Same quantum (same exponent)
print(f"d1.same_quantum(d2): {d1.same_quantum(d2)}")
print(f"d2.same_quantum(d3): {d2.same_quantum(d3)}")

# Min and Max
numbers = [Decimal('10.5'), Decimal('3.2'), Decimal('15.7'), Decimal('1.1')]
print(f"Numbers: {numbers}")
print(f"Min: {min(numbers)}")
print(f"Max: {max(numbers)}")
```

Output:

```
=== Decimal Comparison Methods ===
d1 == d2: True
d1 is d2: False
d1.compare(d2): 0
d1.compare(Decimal('5')): 1
d1.compare(Decimal('15')): -1
d1.same_quantum(d2): True
d2.same_quantum(d3): True
Numbers: [Decimal('10.5'), Decimal('3.2'), Decimal('15.7'), Decimal('1.1')]
Min: 1.1
Max: 15.7
```

Mathematical Functions

```
print("=== Decimal Mathematical Functions ===")

# Square root
num = Decimal('25')
print(f"sqrt({num}) = {num.sqrt()}")

# More complex square root
num2 = Decimal('2')
sqrt_2 = num2.sqrt()
```

```
print(f"sqrt(2) = {sqrt_2}")

# Power operations
base = Decimal('2')
exponent = Decimal('3')
result = base ** exponent
print(f"{base}^{exponent} = {result}")

# Natural exponential and logarithm
from decimal import localcontext
import decimal

# Set higher precision for mathematical operations
with localcontext() as ctx:
    ctx.prec = 50

    # Exponential
    x = Decimal('1')
    try:
        exp_x = x.exp()
        print(f"e^1 = {exp_x}")
    except AttributeError:
        print("exp() method not available in this Python version")

    # Logarithm
    try:
        ln_10 = Decimal('10').ln()
        print(f"ln(10) = {ln_10}")
    except AttributeError:
        print("ln() method not available in this Python version")

# Copy operations
original = Decimal('123.456')
copied = original.copy_abs()
copied_neg = original.copy_negate()

print(f"Original: {original}")
print(f"Copy absolute: {copied}")
print(f"Copy negate: {copied_neg}")
```

Output:

```
=== Decimal Mathematical Functions ===
sqrt(25) = 5
sqrt(2) = 1.414213562373095048801688724
2^3 = 8
e^1 = 2.7182818284590452353602874713526624977572470937
ln(10) = 2.3025850929940456840179914546843642076011014886
Original: 123.456
Copy absolute: 123.456
Copy negate: -123.456
```

10.5 A Decimal Class Application

Let's build a practical application that demonstrates the power of the Decimal class: a loan payment calculator that requires precise financial calculations.

Monthly Payment Calculator

```
from decimal import Decimal, getcontext, ROUND_HALF_UP

# Set precision for financial calculations
getcontext().prec = 10

class LoanCalculator:
    """A precise loan calculator using Decimal for financial accuracy"""

    def __init__(self, principal, annual_rate, years):
        """
        Initialize loan calculator

        Args:
            principal: Loan amount as string or Decimal
            annual_rate: Annual interest rate as decimal (e.g., 0.05 for 5%)
            years: Loan term in years
        """
        self.principal = Decimal(str(principal))
        self.annual_rate = Decimal(str(annual_rate))
        self.years = int(years)
        self.monthly_rate = self.annual_rate / Decimal('12')
        self.total_payments = self.years * 12

    def calculate_monthly_payment(self):
        """Calculate monthly payment using precise decimal arithmetic"""
        if self.annual_rate == 0:
            return self.principal / Decimal(str(self.total_payments))

        # Monthly payment formula: P * [r(1+r)^n] / [(1+r)^n - 1]
        r = self.monthly_rate
        n = Decimal(str(self.total_payments))

        factor = (Decimal('1') + r) ** n
        monthly_payment = self.principal * (r * factor) / (factor - Decimal('1'))

        # Round to cents
        return monthly_payment.quantize(Decimal('0.01'), rounding=ROUND_HALF_UP)

    def generate_amortization_schedule(self):
        """Generate first few payments of amortization schedule"""
        monthly_payment = self.calculate_monthly_payment()
        balance = self.principal

        schedule = []
        for payment_num in range(1, min(13, self.total_payments + 1)):
            interest_payment = (balance * self.monthly_rate).quantize(
                Decimal('0.01'), rounding=ROUND_HALF_UP)
            principal_payment = monthly_payment - interest_payment
```

```
            balance -= principal_payment

            schedule.append({
                'payment': payment_num,
                'monthly_payment': monthly_payment,
                'interest': interest_payment,
                'principal': principal_payment,
                'balance': balance.quantize(Decimal('0.01'), rounding=ROUND_HALF_UP)
            })

        return schedule

    def total_interest_paid(self):
        """Calculate total interest over the life of the loan"""
        monthly_payment = self.calculate_monthly_payment()
        total_paid = monthly_payment * Decimal(str(self.total_payments))
        return (total_paid - self.principal).quantize(Decimal('0.01'))

# Example usage
print("=== Loan Calculator Application ===")

# Create loan scenarios
loans = [
    {"principal": "250000", "rate": "0.045", "years": 30, "description": "30-year mortgag
    {"principal": "25000", "rate": "0.0675", "years": 5, "description": "5-year car loan
    {"principal": "50000", "rate": "0", "years": 10, "description": "Interest-free loan"}
]

for loan_data in loans:
    print(f"\n{loan_data['description']}:")
    print("-" * 50)

    calc = LoanCalculator(
        loan_data["principal"],
        loan_data["rate"],
        loan_data["years"]
    )

    monthly = calc.calculate_monthly_payment()
    total_interest = calc.total_interest_paid()

    print(f"Loan Amount: ${calc.principal:,}")
    print(f"Monthly Payment: ${monthly}")
    print(f"Total Interest: ${total_interest:,}")
    print(f"Total Cost: ${calc.principal + total_interest:,}")

    # Show first 3 payments
    schedule = calc.generate_amortization_schedule()
    print(f"\nFirst 3 Payments:")
    print(f"{'Pmt':&lt;3} {'Payment':&lt;10} {'Interest':&lt;10} {'Principal':&lt;10} {'E
    print("-" * 50)

    for payment in schedule[:3]:
        print(f"{payment['payment']:&lt;3} "
              f"${payment['monthly_payment']:&lt;9} "
              f"${payment['interest']:&lt;9} "
```

```
              f"${payment['principal']:&lt;9} "
              f"${payment['balance']:&lt;11,}")
```

Output:

```
=== Loan Calculator Application ===

30-year mortgage at 4.5%:
--------------------------------------------------
Loan Amount: $250,000
Monthly Payment: $1267.32
Total Interest: $206,235.20
Total Cost: $456,235.20

First 3 Payments:
Pmt Payment    Interest   Principal  Balance
--------------------------------------------------
1   $1267.32   $937.50    $329.82    $249,670.18
2   $1267.32   $936.26    $331.06    $249,339.12
3   $1267.32   $935.02    $332.30    $249,006.82

5-year car loan at 6.75%:
--------------------------------------------------
Loan Amount: $25,000
Monthly Payment: $495.03
Total Interest: $4,701.80
Total Cost: $29,701.80

First 3 Payments:
Pmt Payment    Interest   Principal  Balance
--------------------------------------------------
1   $495.03    $140.63    $354.40    $24,645.60
2   $495.03    $138.63    $356.40    $24,289.20
3   $495.03    $136.62    $358.41    $23,930.79

Interest-free loan:
--------------------------------------------------
Loan Amount: $50,000
Monthly Payment: $416.67
Total Interest: $0.00
Total Cost: $50,000.00

First 3 Payments:
Pmt Payment    Interest   Principal  Balance
--------------------------------------------------
1   $416.67    $0.00      $416.67    $49,583.33
2   $416.67    $0.00      $416.67    $49,166.66
3   $416.67    $0.00      $416.67    $48,749.99
```

10.6 Designing a Money Class

A Money class encapsulates currency amounts with their associated currency codes, providing a foundation for international financial applications.

Money Class Design Principles

```
from decimal import Decimal, ROUND_HALF_UP
from typing import Union, Optional

class InvalidCurrencyError(ValueError):
    """Raised when an invalid currency code is used"""
    pass

class IncompatibleCurrencyError(ValueError):
    """Raised when operations are attempted between incompatible currencies"""
    pass

# Currency validation
VALID_CURRENCIES = {
    'USD': {'name': 'US Dollar', 'symbol': '$', 'decimals': 2},
    'EUR': {'name': 'Euro', 'symbol': '€', 'decimals': 2},
    'JPY': {'name': 'Japanese Yen', 'symbol': '¥', 'decimals': 0},
    'GBP': {'name': 'British Pound', 'symbol': '£', 'decimals': 2},
    'CHF': {'name': 'Swiss Franc', 'symbol': 'CHF', 'decimals': 2},
    'CAD': {'name': 'Canadian Dollar', 'symbol': 'C$', 'decimals': 2}
}

class Money:
    """
    Represents a monetary amount with currency

    Design principles:
    - Immutable objects
    - Precise decimal arithmetic
    - Currency validation
    - Proper rounding for each currency
    """

    def __init__(self, amount: Union[str, int, float, Decimal], currency: str):
        """
        Initialize a Money object

        Args:
            amount: The monetary amount
            currency: ISO currency code (e.g., 'USD', 'EUR')
        """
        if currency not in VALID_CURRENCIES:
            raise InvalidCurrencyError(f"Invalid currency code: {currency}")

        self._currency = currency
        self._amount = Decimal(str(amount))
        self._currency_info = VALID_CURRENCIES[currency]

        # Round to appropriate decimal places for the currency
```

```
        decimal_places = self._currency_info['decimals']
        if decimal_places &gt; 0:
            quantizer = Decimal('0.' + '0' * (decimal_places - 1) + '1')
        else:
            quantizer = Decimal('1')

        self._amount = self._amount.quantize(quantizer, rounding=ROUND_HALF_UP)

    @property
    def amount(self) -&gt; Decimal:
        """Get the amount as a Decimal"""
        return self._amount

    @property
    def currency(self) -&gt; str:
        """Get the currency code"""
        return self._currency

    def __str__(self) -&gt; str:
        """String representation for display"""
        symbol = self._currency_info['symbol']
        if self._currency_info['decimals'] &gt; 0:
            return f"{symbol}{self._amount}"
        else:
            return f"{symbol}{int(self._amount)}"

    def __repr__(self) -&gt; str:
        """String representation for debugging"""
        return f"Money('{self._amount}', '{self._currency}')"

    def __eq__(self, other) -&gt; bool:
        """Check equality"""
        if not isinstance(other, Money):
            return False
        return (self._amount == other._amount and
                self._currency == other._currency)

    def __hash__(self) -&gt; int:
        """Make Money objects hashable"""
        return hash((self._amount, self._currency))

# Test the basic Money class
print("=== Money Class Design ===")

# Create Money objects
usd_100 = Money('100.00', 'USD')
eur_85 = Money('85.50', 'EUR')
yen_10000 = Money('10000', 'JPY')

print(f"USD Money: {usd_100}")
print(f"EUR Money: {eur_85}")
print(f"JPY Money: {yen_10000}")
print(f"USD Repr: {repr(usd_100)}")

# Test currency validation
try:
```

```
    invalid_money = Money('100', 'XYZ')
except InvalidCurrencyError as e:
    print(f"Currency validation error: {e}")

# Test equality
usd_100_copy = Money('100.00', 'USD')
print(f"usd_100 == usd_100_copy: {usd_100 == usd_100_copy}")
print(f"usd_100 == eur_85: {usd_100 == eur_85}")

# Test automatic rounding
precise_usd = Money('123.456789', 'USD')
precise_jpy = Money('123.456789', 'JPY')
print(f"USD with precision: {precise_usd}")
print(f"JPY with precision: {precise_jpy}")
```

Output:

```
=== Money Class Design ===
USD Money: $100.00
EUR Money: €85.50
JPY Money: ¥10000
USD Repr: Money('100.00', 'USD')
Currency validation error: Invalid currency code: XYZ
usd_100 == usd_100_copy: True
usd_100 == eur_85: False
USD with precision: $123.46
JPY with precision: ¥123
```

10.7 Writing the Basic Money Class (Containment)

The Money class uses containment to encapsulate a Decimal object, providing a clean interface while leveraging Decimal's precision.

Complete Money Class with Arithmetic Operations

```
from decimal import Decimal, ROUND_HALF_UP
from typing import Union

class Money:
    """Complete Money class with arithmetic operations using containment"""

    def __init__(self, amount: Union[str, int, float, Decimal], currency: str = 'USD'):
        """Initialize Money object with amount and currency"""
        if currency not in VALID_CURRENCIES:
            raise InvalidCurrencyError(f"Invalid currency: {currency}")

        self._currency = currency
        # Containment: Money contains a Decimal object
        self._amount = Decimal(str(amount))
        self._currency_info = VALID_CURRENCIES[currency]

        # Round to currency precision
```

```
        decimal_places = self._currency_info['decimals']
        if decimal_places &gt; 0:
            quantizer = Decimal('0.' + '0' * (decimal_places - 1) + '1')
            self._amount = self._amount.quantize(quantizer, rounding=ROUND_HALF_UP)

    def __add__(self, other):
        """Add two Money objects or Money and numeric value"""
        if isinstance(other, Money):
            if self._currency != other._currency:
                raise IncompatibleCurrencyError(
                    f"Cannot add {self._currency} and {other._currency}")
            return Money(self._amount + other._amount, self._currency)
        else:
            # Add numeric value
            return Money(self._amount + Decimal(str(other)), self._currency)

    def __radd__(self, other):
        """Right addition for numeric + Money"""
        return self.__add__(other)

    def __sub__(self, other):
        """Subtract Money objects or numeric values"""
        if isinstance(other, Money):
            if self._currency != other._currency:
                raise IncompatibleCurrencyError(
                    f"Cannot subtract {other._currency} from {self._currency}")
            return Money(self._amount - other._amount, self._currency)
        else:
            return Money(self._amount - Decimal(str(other)), self._currency)

    def __rsub__(self, other):
        """Right subtraction for numeric - Money"""
        return Money(Decimal(str(other)) - self._amount, self._currency)

    def __mul__(self, other):
        """Multiply Money by a numeric value"""
        if isinstance(other, Money):
            raise TypeError("Cannot multiply Money by Money")
        return Money(self._amount * Decimal(str(other)), self._currency)

    def __rmul__(self, other):
        """Right multiplication for numeric * Money"""
        return self.__mul__(other)

    def __truediv__(self, other):
        """Divide Money by numeric value or get ratio of two Money objects"""
        if isinstance(other, Money):
            if self._currency != other._currency:
                raise IncompatibleCurrencyError(
                    f"Cannot divide {self._currency} by {other._currency}")
            return self._amount / other._amount  # Returns Decimal ratio
        else:
            return Money(self._amount / Decimal(str(other)), self._currency)

    def __floordiv__(self, other):
        """Floor division"""
```

```
        if isinstance(other, Money):
            if self._currency != other._currency:
                raise IncompatibleCurrencyError(
                    f"Cannot divide {self._currency} by {other._currency}")
            return self._amount // other._amount
        else:
            return Money(self._amount // Decimal(str(other)), self._currency)

    def __mod__(self, other):
        """Modulo operation"""
        if isinstance(other, Money):
            if self._currency != other._currency:
                raise IncompatibleCurrencyError(
                    f"Cannot mod {self._currency} by {other._currency}")
            return Money(self._amount % other._amount, self._currency)
        else:
            return Money(self._amount % Decimal(str(other)), self._currency)

    def __neg__(self):
        """Unary minus"""
        return Money(-self._amount, self._currency)

    def __abs__(self):
        """Absolute value"""
        return Money(abs(self._amount), self._currency)

    def __lt__(self, other):
        """Less than comparison"""
        if not isinstance(other, Money):
            raise TypeError("Cannot compare Money with non-Money")
        if self._currency != other._currency:
            raise IncompatibleCurrencyError(
                f"Cannot compare {self._currency} with {other._currency}")
        return self._amount &lt; other._amount

    def __le__(self, other):
        """Less than or equal comparison"""
        return self &lt; other or self == other

    def __gt__(self, other):
        """Greater than comparison"""
        if not isinstance(other, Money):
            raise TypeError("Cannot compare Money with non-Money")
        if self._currency != other._currency:
            raise IncompatibleCurrencyError(
                f"Cannot compare {self._currency} with {other._currency}")
        return self._amount &gt; other._amount

    def __ge__(self, other):
        """Greater than or equal comparison"""
        return self &gt; other or self == other

    # Properties to access contained Decimal
    @property
    def amount(self):
        """Get the contained Decimal amount"""
```

```
        return self._amount

    @property
    def currency(self):
        """Get the currency code"""
        return self._currency

    def __str__(self):
        """String representation"""
        symbol = self._currency_info['symbol']
        if self._currency_info['decimals'] &gt; 0:
            return f"{symbol}{self._amount}"
        else:
            return f"{symbol}{int(self._amount)}"

    def __repr__(self):
        """Debug representation"""
        return f"Money({self._amount}, '{self._currency}')"

# Demonstrate arithmetic operations
print("=== Money Class Arithmetic Operations ===")

# Create Money objects
price1 = Money('19.95', 'USD')
price2 = Money('5.00', 'USD')
tax_rate = Decimal('0.08')

print(f"Price 1: {price1}")
print(f"Price 2: {price2}")

# Addition
subtotal = price1 + price2
print(f"Subtotal: {price1} + {price2} = {subtotal}")

# Multiplication
tax = subtotal * tax_rate
print(f"Tax (8%): {subtotal} * {tax_rate} = {tax}")

# Final total
total = subtotal + tax
print(f"Total: {subtotal} + {tax} = {total}")

# Division
unit_price = total / 3
print(f"Unit price (total / 3): {unit_price}")

# Comparison
expensive_item = Money('100.00', 'USD')
cheap_item = Money('5.00', 'USD')

print(f"\n=== Comparisons ===")
print(f"{expensive_item} &gt; {cheap_item}: {expensive_item &gt; cheap_item}")
print(f"{cheap_item} &lt; {expensive_item}: {cheap_item &lt; expensive_item}")

# Error handling
print(f"\n=== Error Handling ===")
```

```
eur_price = Money('20.00', 'EUR')

try:
    mixed_sum = price1 + eur_price
except IncompatibleCurrencyError as e:
    print(f"Currency error: {e}")

try:
    invalid_comparison = price1 &gt; "not money"
except TypeError as e:
    print(f"Type error: {e}")
```

Output:

```
=== Money Class Arithmetic Operations ===
Price 1: $19.95
Price 2: $5.00
Subtotal: $19.95 + $5.00 = $24.95
Tax (8%): $24.95 * 0.08 = $2.00
Total: $24.95 + $2.00 = $26.95
Unit price (total / 3): $8.98

=== Comparisons ===
$100.00 &gt; $5.00: True
$5.00 &lt; $100.00: True

=== Error Handling ===
Currency error: Cannot add USD and EUR
Type error: Cannot compare Money with non-Money
```

10.8 Displaying Money Objects (str, repr)

Proper string representation is crucial for Money objects, providing both user-friendly display and debugging information.

Enhanced String Representation

```
class EnhancedMoney(Money):
    """Enhanced Money class with advanced string formatting"""

    def __init__(self, amount, currency='USD', locale=None):
        super().__init__(amount, currency)
        self._locale = locale or 'en_US'

    def __str__(self):
        """User-friendly string representation"""
        symbol = self._currency_info['symbol']
        decimal_places = self._currency_info['decimals']

        if decimal_places &gt; 0:
            # Format with thousands separator for readability
            if abs(self._amount) &gt;= 1000:
```

```
                return f"{symbol}{self._amount:,}"
            else:
                return f"{symbol}{self._amount}"
        else:
            # No decimal places (like JPY)
            amount_int = int(self._amount)
            if abs(amount_int) &gt;= 1000:
                return f"{symbol}{amount_int:,}"
            else:
                return f"{symbol}{amount_int}"

    def __repr__(self):
        """Developer-friendly representation"""
        return (f"Money(amount={self._amount}, currency='{self._currency}', "
                f"locale='{self._locale}')")

    def __format__(self, format_spec):
        """Custom formatting support"""
        if format_spec == 'code':
            # Show currency code instead of symbol
            return f"{self._amount} {self._currency}"
        elif format_spec == 'full':
            # Full name format
            currency_name = self._currency_info['name']
            return f"{self._amount} {currency_name}"
        elif format_spec == 'accounting':
            # Accounting format with parentheses for negative
            symbol = self._currency_info['symbol']
            if self._amount &lt; 0:
                return f"({symbol}{abs(self._amount)})"
            else:
                return f"{symbol}{self._amount}"
        else:
            # Default format
            return str(self)

    def to_words(self):
        """Convert money amount to words (simplified)"""
        # This is a simplified version - real implementation would be more complex
        units = ["", "thousand", "million", "billion"]

        amount = abs(self._amount)
        integer_part = int(amount)
        decimal_part = int((amount % 1) * 100)

        def number_to_words(n):
            if n == 0:
                return "zero"

            ones = ["", "one", "two", "three", "four", "five", "six", "seven", "eight", '
            teens = ["ten", "eleven", "twelve", "thirteen", "fourteen", "fifteen",
                    "sixteen", "seventeen", "eighteen", "nineteen"]
            tens = ["", "", "twenty", "thirty", "forty", "fifty", "sixty", "seventy", "ei

            if n &lt; 10:
                return ones[n]
```

```
            elif n &lt; 20:
                return teens[n-10]
            elif n &lt; 100:
                return tens[n//10] + ("" if n%10 == 0 else "-" + ones[n%10])
            elif n &lt; 1000:
                return ones[n//100] + " hundred" + ("" if n%100 == 0 else " " + number_to
            else:
                for i, unit in enumerate(units):
                    if n &lt; 1000 ** (i + 1):
                        return (number_to_words(n // (1000**i)) + " " + units[i] +
                               ("" if n % (1000**i) == 0 else " " + number_to_words(n % (

        result = ""
        if self._amount &lt; 0:
            result += "negative "

        result += number_to_words(integer_part) + f" {self._currency}"

        if decimal_part &gt; 0 and self._currency_info['decimals'] &gt; 0:
            result += f" and {decimal_part} cents"

        return result.strip()

# Demonstration of enhanced formatting
print("=== Enhanced Money Formatting ===")

# Create various money amounts
amounts = [
    EnhancedMoney('1234.56', 'USD'),
    EnhancedMoney('1000000.00', 'USD'),
    EnhancedMoney('-500.75', 'USD'),
    EnhancedMoney('999', 'JPY'),
    EnhancedMoney('50.00', 'EUR')
]

for money in amounts:
    print(f"Amount: {money}")
    print(f"  __repr__(): {repr(money)}")
    print(f"  Code format: {money:code}")
    print(f"  Full format: {money:full}")
    print(f"  Accounting format: {money:accounting}")
    if money.currency == 'USD' and abs(money.amount) &lt;= 999:
        print(f"  In words: {money.to_words()}")
    print()

# Custom Money display class
class DisplayableMoney(Money):
    """Money class with multiple display options"""

    def display_table_row(self, description="", width=40):
        """Format for table display"""
        desc = description[:width-15].ljust(width-15)
        amount_str = str(self).rjust(12)
        return f"{desc} {amount_str}"

    def display_receipt_line(self, description="", quantity=1):
```

```
        """Format for receipt display"""
        if quantity == 1:
            return f"{description:&lt;30} {str(self):&gt;10}"
        else:
            unit_price = self / quantity
            return f"{description:&lt;20} {quantity:&gt;3} × {unit_price} = {str(self):&g

# Receipt example
print("=== Receipt Formatting ===")
items = [
    (DisplayableMoney('12.99', 'USD'), "Coffee Beans", 2),
    (DisplayableMoney('4.50', 'USD'), "Pastry", 1),
    (DisplayableMoney('15.00', 'USD'), "Sandwich", 3)
]

print("COFFEE SHOP RECEIPT")
print("=" * 45)

subtotal = Money('0.00', 'USD')
for total_price, description, qty in items:
    print(total_price.display_receipt_line(description, qty))
    subtotal += total_price

print("-" * 45)
tax = subtotal * Decimal('0.08')
total = subtotal + tax

subtotal_display = DisplayableMoney(subtotal.amount, 'USD')
tax_display = DisplayableMoney(tax.amount, 'USD')
total_display = DisplayableMoney(total.amount, 'USD')

print(subtotal_display.display_table_row("Subtotal"))
print(tax_display.display_table_row("Tax (8%)"))
print("=" * 45)
print(total_display.display_table_row("TOTAL"))
```

Output:

```
=== Enhanced Money Formatting ===
Amount: $1,234.56
  __repr__(): Money(amount=1234.56, currency='USD', locale='en_US')
  Code format: 1234.56 USD
  Full format: 1234.56 US Dollar
  Accounting format: $1,234.56
  In words: one thousand two hundred thirty-four USD and 56 cents

Amount: $1,000,000.00
  __repr__(): Money(amount=1000000.00, currency='USD', locale='en_US')
  Code format: 1000000.00 USD
  Full format: 1000000.00 US Dollar
  Accounting format: $1,000,000.00

Amount: ($500.75)
  __repr__(): Money(amount=-500.75, currency='USD', locale='en_US')
  Code format: -500.75 USD
```

```
  Full format: -500.75 US Dollar
  Accounting format: ($500.75)

Amount: ¥999
  __repr__(): Money(amount=999, currency='JPY', locale='en_US')
  Code format: 999 JPY
  Full format: 999 Japanese Yen
  Accounting format: ¥999
  In words: nine hundred ninety-nine JPY

Amount: €50.00
  __repr__(): Money(amount=50.00, currency='EUR', locale='en_US')
  Code format: 50.00 EUR
  Full format: 50.00 Euro
  Accounting format: €50.00
  In words: fifty EUR and 0 cents

=== Receipt Formatting ===
COFFEE SHOP RECEIPT
=============================================
Coffee Beans          2 × $6.50 =     $12.99
Pastry                                  $4.50
Sandwich              3 × $5.00 =     $15.00
---------------------------------------------
Subtotal                              $32.49
Tax (8%)                               $2.60
=============================================
TOTAL                                 $35.09
```

10.9 Other Monetary Operations

Beyond basic arithmetic, Money objects need specialized operations for real-world financial applications.

Advanced Money Operations

```
from decimal import Decimal, ROUND_HALF_UP, ROUND_DOWN, ROUND_UP
import operator

class AdvancedMoney(Money):
    """Money class with advanced financial operations"""

    def split(self, ways: int, rounding=ROUND_HALF_UP):
        """Split money amount into equal parts"""
        if ways &lt;= 0:
            raise ValueError("Cannot split into zero or negative parts")

        quotient, remainder = divmod(self._amount, ways)

        # Create base amounts
        amounts = [Money(quotient, self._currency) for _ in range(ways)]

        # Distribute remainder
```

```
        remainder_cents = int(remainder * (10 ** self._currency_info['decimals']))
        for i in range(remainder_cents):
            amounts[i] += Money(Decimal('0.01'), self._currency)

        return amounts

    def allocate(self, ratios):
        """Allocate money according to given ratios"""
        if not ratios or all(r &lt;= 0 for r in ratios):
            raise ValueError("All ratios must be positive")

        total_ratio = sum(ratios)
        allocated = []
        remaining = self._amount

        for i, ratio in enumerate(ratios):
            if i == len(ratios) - 1:  # Last allocation gets remainder
                allocated.append(Money(remaining, self._currency))
            else:
                amount = (self._amount * Decimal(str(ratio)) /
                         Decimal(str(total_ratio))).quantize(
                    Decimal('0.01'), rounding=ROUND_DOWN)
                allocated.append(Money(amount, self._currency))
                remaining -= amount

        return allocated

    def compound_interest(self, rate: Decimal, periods: int):
        """Calculate compound interest"""
        if periods &lt;= 0:
            return Money('0', self._currency)

        rate_decimal = Decimal(str(rate))
        final_amount = self._amount * ((1 + rate_decimal) ** periods)
        return Money(final_amount - self._amount, self._currency)

    def present_value(self, rate: Decimal, periods: int):
        """Calculate present value"""
        if periods &lt;= 0:
            return Money(self._amount, self._currency)

        rate_decimal = Decimal(str(rate))
        pv_amount = self._amount / ((1 + rate_decimal) ** periods)
        return Money(pv_amount, self._currency)

    def percentage_of(self, total):
        """Calculate what percentage this amount is of a total"""
        if not isinstance(total, Money) or total.currency != self._currency:
            raise IncompatibleCurrencyError("Total must be same currency")

        if total.amount == 0:
            return Decimal('0')

        return (self._amount / total._amount * 100).quantize(Decimal('0.01'))

    @classmethod
```

```
    def sum(cls, money_list):
        """Sum a list of Money objects"""
        if not money_list:
            return None

        result = money_list[^0]
        for money in money_list[1:]:
            result += money
        return result

    @classmethod
    def max(cls, money_list):
        """Find maximum Money object in list"""
        if not money_list:
            return None
        return max(money_list)

    @classmethod
    def min(cls, money_list):
        """Find minimum Money object in list"""
        if not money_list:
            return None
        return min(money_list)

    def apply_discount(self, discount_rate: Decimal):
        """Apply a percentage discount"""
        discount_amount = self * discount_rate
        return self - discount_amount, discount_amount

# Demonstration of advanced operations
print("=== Advanced Money Operations ===")

# Bill splitting
dinner_bill = AdvancedMoney('127.83', 'USD')
print(f"Total dinner bill: {dinner_bill}")

# Split equally among 5 people
split_amounts = dinner_bill.split(5)
print(f"Split 5 ways: {[str(amount) for amount in split_amounts]}")
print(f"Verification: {AdvancedMoney.sum(split_amounts)}")

# Allocation by ratios (investment portfolio)
investment = AdvancedMoney('10000.00', 'USD')
allocation_ratios = [60, 25, 15]  # 60% stocks, 25% bonds, 15% cash
allocated = investment.allocate(allocation_ratios)

print(f"\n=== Investment Allocation ===")
print(f"Total investment: {investment}")
categories = ['Stocks (60%)', 'Bonds (25%)', 'Cash (15%)']
for category, amount in zip(categories, allocated):
    percentage = amount.percentage_of(investment)
    print(f"{category}: {amount} ({percentage}%)")

# Compound interest calculation
principal = AdvancedMoney('1000.00', 'USD')
annual_rate = Decimal('0.05')  # 5% per year
```

```
years = 10

interest_earned = principal.compound_interest(annual_rate, years)
final_value = principal + interest_earned

print(f"\n=== Compound Interest ===")
print(f"Principal: {principal}")
print(f"Rate: {annual_rate * 100}% per year")
print(f"Years: {years}")
print(f"Interest earned: {interest_earned}")
print(f"Final value: {final_value}")

# Present value calculation
future_value = AdvancedMoney('1500.00', 'USD')
pv = future_value.present_value(annual_rate, years)
print(f"Present value of {future_value} in {years} years at {annual_rate * 100}%: {pv}")

# Discount application
original_price = AdvancedMoney('299.99', 'USD')
discount_rate = Decimal('0.20')  # 20% discount

discounted_price, discount_amount = original_price.apply_discount(discount_rate)
print(f"\n=== Discount Application ===")
print(f"Original price: {original_price}")
print(f"Discount (20%): -{discount_amount}")
print(f"Final price: {discounted_price}")

# Working with lists of Money objects
expenses = [
    AdvancedMoney('45.67', 'USD'),
    AdvancedMoney('123.89', 'USD'),
    AdvancedMoney('67.23', 'USD'),
    AdvancedMoney('234.56', 'USD'),
    AdvancedMoney('89.01', 'USD')
]

print(f"\n=== Expense Analysis ===")
print(f"Individual expenses: {[str(exp) for exp in expenses]}")
print(f"Total expenses: {AdvancedMoney.sum(expenses)}")
print(f"Highest expense: {AdvancedMoney.max(expenses)}")
print(f"Lowest expense: {AdvancedMoney.min(expenses)}")

# Percentage analysis
total_expenses = AdvancedMoney.sum(expenses)
print(f"\nExpense breakdown:")
for i, expense in enumerate(expenses, 1):
    percentage = expense.percentage_of(total_expenses)
    print(f"Expense {i}: {expense} ({percentage}%)")
```

Output:

```
=== Advanced Money Operations ===
Total dinner bill: $127.83
Split 5 ways: ['$25.57', '$25.57', '$25.56', '$25.56', '$25.57']
Verification: $127.83
```

```
=== Investment Allocation ===
Total investment: $10,000.00
Stocks (60%): $6,000.00 (60.00%)
Bonds (25%): $2,500.00 (25.00%)
Cash (15%): $1,500.00 (15.00%)

=== Compound Interest ===
Principal: $1,000.00
Rate: 5.00% per year
Years: 10
Interest earned: $628.89
Final value: $1,628.89
Present value of $1,500.00 in 10 years at 5.00%: $921.71

=== Discount Application ===
Original price: $299.99
Discount (20%): -$60.00
Final price: $239.99

=== Expense Analysis ===
Individual expenses: ['$45.67', '$123.89', '$67.23', '$234.56', '$89.01']
Total expenses: $560.36
Highest expense: $234.56
Lowest expense: $45.67

Expense breakdown:
Expense 1: $45.67 (8.15%)
Expense 2: $123.89 (22.11%)
Expense 3: $67.23 (12.00%)
Expense 4: $234.56 (41.86%)
Expense 5: $89.01 (15.89%)
```

10.10 Demo: A Money Calculator

Let's build a comprehensive money calculator that demonstrates all the concepts we've learned.

Interactive Money Calculator

```
from decimal import Decimal, getcontext
import re

# Set high precision for calculations
getcontext().prec = 28

class MoneyCalculator:
    """Interactive calculator for money operations"""

    def __init__(self):
        self.memory = {}
        self.history = []
        self.default_currency = 'USD'
```

```
    def parse_money_input(self, input_str):
        """Parse money input like '$100.50 USD' or '100.50' """
        # Remove whitespace
        input_str = input_str.strip()

        # Pattern to match: [symbol]amount [currency]
        pattern = r'([£$€¥]?)([0-9,]+\.?[0-9]*)\s*([A-Z]{3})?'
        match = re.match(pattern, input_str)

        if not match:
            raise ValueError(f"Invalid money format: {input_str}")

        symbol, amount_str, currency = match.groups()

        # Remove commas from amount
        amount_str = amount_str.replace(',', '')

        # Determine currency from symbol if not specified
        if currency is None:
            symbol_to_currency = {'$': 'USD', '€': 'EUR', '£': 'GBP', '¥': 'JPY'}
            currency = symbol_to_currency.get(symbol, self.default_currency)

        return AdvancedMoney(amount_str, currency)

    def calculate(self, expression):
        """Calculate money expression"""
        try:
            # Simple calculator - in real implementation, you'd use a proper parser
            self.history.append(expression)

            # Handle basic operations
            if '+' in expression:
                parts = expression.split('+')
                if len(parts) == 2:
                    left = self.parse_money_input(parts[^0].strip())
                    right = self.parse_money_input(parts[^1].strip())
                    return left + right

            elif '-' in expression:
                parts = expression.split('-')
                if len(parts) == 2:
                    left = self.parse_money_input(parts[^0].strip())
                    right = self.parse_money_input(parts[^1].strip())
                    return left - right

            elif '*' in expression:
                parts = expression.split('*')
                if len(parts) == 2:
                    money_part = self.parse_money_input(parts[^0].strip())
                    multiplier = Decimal(parts[^1].strip())
                    return money_part * multiplier

            elif '/' in expression:
                parts = expression.split('/')
                if len(parts) == 2:
                    money_part = self.parse_money_input(parts[^0].strip())
```

```
                    divisor = Decimal(parts[^1].strip())
                    return money_part / divisor

            else:
                # Single money value
                return self.parse_money_input(expression)

        except Exception as e:
            return f"Error: {e}"

    def store_memory(self, name, value):
        """Store value in memory"""
        if isinstance(value, AdvancedMoney):
            self.memory[name] = value
            return f"Stored {value} as {name}"
        else:
            return "Can only store Money objects in memory"

    def recall_memory(self, name):
        """Recall value from memory"""
        return self.memory.get(name, "Not found in memory")

    def show_memory(self):
        """Show all memory contents"""
        if not self.memory:
            return "Memory is empty"

        result = "Memory contents:\n"
        for name, value in self.memory.items():
            result += f"  {name}: {value}\n"
        return result.strip()

    def show_history(self):
        """Show calculation history"""
        if not self.history:
            return "No calculation history"

        return "Calculation history:\n" + "\n".join(f"  {i+1}. {expr}"
                                                    for i, expr in enumerate(self.history[

# Demo the Money Calculator
print("=== Money Calculator Demo ===")

calc = MoneyCalculator()

# Test calculations
test_expressions = [
    "$100.00 USD + $50.00 USD",
    "€200.50 EUR - €75.25 EUR",
    "$1,234.56 USD * 1.08",
    "$500.00 USD / 4",
    "¥10000 JPY + ¥5000 JPY"
]

for expression in test_expressions:
    result = calc.calculate(expression)
```

```
    print(f"{expression} = {result}")

print()

# Memory operations
savings = calc.parse_money_input("$5,000.00 USD")
checking = calc.parse_money_input("$1,500.75 USD")

print(calc.store_memory("savings", savings))
print(calc.store_memory("checking", checking))
print()
print(calc.show_memory())
print()

# Advanced calculations
print("=== Advanced Calculator Features ===")

# Bill splitting scenario
total_bill = calc.parse_money_input("$247.83 USD")
tip_rate = Decimal('0.18')  # 18% tip
number_of_people = 6

bill_with_tip = total_bill * (1 + tip_rate)
per_person = bill_with_tip / number_of_people

print(f"Restaurant Bill Calculator:")
print(f"  Subtotal: {total_bill}")
print(f"  Tip (18%): {total_bill * tip_rate}")
print(f"  Total with tip: {bill_with_tip}")
print(f"  Split {number_of_people} ways: {per_person} per person")

# Investment calculator
print(f"\nInvestment Calculator:")
initial_investment = calc.parse_money_input("$10,000.00 USD")
monthly_contribution = calc.parse_money_input("$500.00 USD")
annual_rate = Decimal('0.07')  # 7% annual return
years = 10

# Simple compound interest calculation
final_value = initial_investment
for year in range(years):
    # Add monthly contributions
    final_value += monthly_contribution * 12
    # Apply annual interest
    final_value += final_value * annual_rate

interest_earned = final_value - initial_investment - (monthly_contribution * 12 * years)

print(f"  Initial investment: {initial_investment}")
print(f"  Monthly contribution: {monthly_contribution}")
print(f"  Annual return: {annual_rate * 100}%")
print(f"  Time period: {years} years")
print(f"  Total contributions: {initial_investment + (monthly_contribution * 12 * years)}
print(f"  Interest earned: {interest_earned}")
print(f"  Final value: {final_value}")
```

```
# Currency conversion example (simplified)
print(f"\nCurrency Conversion Example:")

# Mock exchange rates (in real app, these would be fetched from an API)
exchange_rates = {
    ('USD', 'EUR'): Decimal('0.85'),
    ('USD', 'GBP'): Decimal('0.75'),
    ('USD', 'JPY'): Decimal('110.0'),
    ('EUR', 'USD'): Decimal('1.18'),
    ('GBP', 'USD'): Decimal('1.33'),
    ('JPY', 'USD'): Decimal('0.009')
}

def convert_currency(money, target_currency):
    """Simple currency conversion"""
    if money.currency == target_currency:
        return money

    rate_key = (money.currency, target_currency)
    if rate_key in exchange_rates:
        rate = exchange_rates[rate_key]
        new_amount = money.amount * rate
        return AdvancedMoney(new_amount, target_currency)
    else:
        return f"No exchange rate available for {money.currency} to {target_currency}"

usd_amount = calc.parse_money_input("$1,000.00 USD")
eur_converted = convert_currency(usd_amount, 'EUR')
gbp_converted = convert_currency(usd_amount, 'GBP')
jpy_converted = convert_currency(usd_amount, 'JPY')

print(f"  {usd_amount} converts to:")
print(f"    EUR: {eur_converted}")
print(f"    GBP: {gbp_converted}")
print(f"    JPY: {jpy_converted}")
```

Output:

```
=== Money Calculator Demo ===
$100.00 USD + $50.00 USD = $150.00
€200.50 EUR - €75.25 EUR = €125.25
$1,234.56 USD * 1.08 = $1,333.32
$500.00 USD / 4 = $125.00
¥10000 JPY + ¥5000 JPY = ¥15000

Stored $5,000.00 as savings
Stored $1,500.75 as checking

Memory contents:
  savings: $5,000.00
  checking: $1,500.75

=== Advanced Calculator Features ===
Restaurant Bill Calculator:
  Subtotal: $247.83
```

```
  Tip (18%): $44.61
  Total with tip: $292.44
  Split 6 ways: $48.74 per person

Investment Calculator:
  Initial investment: $10,000.00
  Monthly contribution: $500.00
  Annual return: 7.00%
  Time period: 10 years
  Total contributions: $70,000.00
  Interest earned: $34,768.47
  Final value: $104,768.47

Currency Conversion Example:
  $1,000.00 converts to:
    EUR: €850.00
    GBP: £750.00
    JPY: ¥110,000
```

10.11 Setting the Default Currency

Managing default currencies and currency context is important for international applications.

Currency Context Manager

```
from contextlib import contextmanager
from typing import Optional
import threading

class CurrencyContext:
    """Thread-safe currency context manager"""

    def __init__(self):
        self._local = threading.local()
        self._global_default = 'USD'

    def get_default_currency(self) -&gt; str:
        """Get current default currency"""
        return getattr(self._local, 'currency', self._global_default)

    def set_default_currency(self, currency: str):
        """Set default currency for current thread"""
        if currency not in VALID_CURRENCIES:
            raise InvalidCurrencyError(f"Invalid currency: {currency}")
        self._local.currency = currency

    def set_global_default(self, currency: str):
        """Set global default currency"""
        if currency not in VALID_CURRENCIES:
            raise InvalidCurrencyError(f"Invalid currency: {currency}")
        self._global_default = currency

    @contextmanager
```

```python
    def currency_context(self, currency: str):
        """Temporary currency context"""
        old_currency = self.get_default_currency()
        self.set_default_currency(currency)
        try:
            yield currency
        finally:
            if hasattr(self._local, 'currency'):
                self._local.currency = old_currency
            else:
                delattr(self._local, 'currency')

# Global currency context instance
currency_context = CurrencyContext()

class ContextAwareMoney(AdvancedMoney):
    """Money class that uses currency context"""

    def __init__(self, amount, currency=None):
        if currency is None:
            currency = currency_context.get_default_currency()
        super().__init__(amount, currency)

    @classmethod
    def from_string(cls, amount_str: str, currency=None):
        """Create Money from string with optional currency override"""
        return cls(amount_str, currency)

# Demonstrate currency context
print("=== Currency Context Management ===")

# Default behavior
money1 = ContextAwareMoney('100.00')
print(f"Default currency money: {money1}")
print(f"Current default currency: {currency_context.get_default_currency()}")

# Change global default
currency_context.set_global_default('EUR')
money2 = ContextAwareMoney('100.00')
print(f"After setting global default to EUR: {money2}")

# Use context manager for temporary changes
print(f"\nUsing context managers:")

with currency_context.currency_context('GBP'):
    money_gbp = ContextAwareMoney('100.00')
    print(f"Inside GBP context: {money_gbp}")

    # Nested context
    with currency_context.currency_context('JPY'):
        money_jpy = ContextAwareMoney('10000')
        print(f"Inside nested JPY context: {money_jpy}")

    # Back to GBP context
    money_gbp2 = ContextAwareMoney('50.00')
    print(f"Back in GBP context: {money_gbp2}")
```

```
# Outside context - back to global default
money3 = ContextAwareMoney('75.00')
print(f"Outside context (global default): {money3}")

# Application-specific currency settings
class ShoppingCart:
    """Shopping cart with currency context"""

    def __init__(self, store_currency='USD'):
        self.store_currency = store_currency
        self.items = []
        self.tax_rate = Decimal('0.08')

    def add_item(self, description: str, price, quantity: int = 1):
        """Add item to cart"""
        with currency_context.currency_context(self.store_currency):
            if isinstance(price, str):
                money_price = ContextAwareMoney(price)
            else:
                money_price = price

            total_price = money_price * quantity
            self.items.append({
                'description': description,
                'unit_price': money_price,
                'quantity': quantity,
                'total': total_price
            })

    def calculate_total(self):
        """Calculate cart total with tax"""
        subtotal = ContextAwareMoney('0', self.store_currency)

        for item in self.items:
            subtotal += item['total']

        tax = subtotal * self.tax_rate
        total = subtotal + tax

        return {
            'subtotal': subtotal,
            'tax': tax,
            'total': total
        }

    def display_cart(self):
        """Display cart contents"""
        print(f"\nShopping Cart ({self.store_currency}):")
        print("=" * 50)

        for item in self.items:
            if item['quantity'] == 1:
                print(f"{item['description']:&lt;30} {item['total']}")
            else:
                print(f"{item['description']:&lt;20} "
```

```
                    f"{item['quantity']} × {item['unit_price']} = {item['total']}")

        totals = self.calculate_total()
        print("-" * 50)
        print(f"{'Subtotal':&lt;30} {totals['subtotal']}")
        print(f"{'Tax (8%)':&lt;30} {totals['tax']}")
        print("=" * 50)
        print(f"{'TOTAL':&lt;30} {totals['total']}")

# Demonstrate shopping cart with different currencies
print(f"\n=== Multi-Currency Shopping Carts ===")

# US Store
us_cart = ShoppingCart('USD')
us_cart.add_item("Laptop", "999.99")
us_cart.add_item("Mouse", "29.99", 2)
us_cart.add_item("Keyboard", "79.99")
us_cart.display_cart()

# European Store
eu_cart = ShoppingCart('EUR')
eu_cart.add_item("Laptop", "849.99")
eu_cart.add_item("Mouse", "24.99", 2)
eu_cart.add_item("Keyboard", "69.99")
eu_cart.display_cart()

# UK Store
uk_cart = ShoppingCart('GBP')
uk_cart.add_item("Laptop", "749.99")
uk_cart.add_item("Mouse", "19.99", 2)
uk_cart.add_item("Keyboard", "59.99")
uk_cart.display_cart()

# Currency preference manager
class UserCurrencyPreferences:
    """Manage user currency preferences"""

    def __init__(self):
        self.user_preferences = {}

    def set_user_currency(self, user_id: str, preferred_currency: str):
        """Set user's preferred currency"""
        if preferred_currency not in VALID_CURRENCIES:
            raise InvalidCurrencyError(f"Invalid currency: {preferred_currency}")
        self.user_preferences[user_id] = preferred_currency

    def get_user_currency(self, user_id: str) -&gt; str:
        """Get user's preferred currency"""
        return self.user_preferences.get(user_id, currency_context.get_default_currency()

    @contextmanager
    def user_context(self, user_id: str):
        """Create context for specific user"""
        user_currency = self.get_user_currency(user_id)
        with currency_context.currency_context(user_currency):
            yield user_currency
```

```python
# Demonstrate user preferences
print(f"\n=== User Currency Preferences ===")

prefs = UserCurrencyPreferences()
prefs.set_user_currency("alice", "EUR")
prefs.set_user_currency("bob", "GBP")
prefs.set_user_currency("charlie", "JPY")

users = ["alice", "bob", "charlie", "dave"]  # dave has no preference

for user in users:
    with prefs.user_context(user):
        balance = ContextAwareMoney('1000')
        print(f"{user.capitalize()}'s balance: {balance}")
```

Output:

```
=== Currency Context Management ===
Default currency money: $100.00
Current default currency: USD
After setting global default to EUR: €100.00

Using context managers:
Inside GBP context: £100.00
Inside nested JPY context: ¥10000
Back in GBP context: £50.00
Outside context (global default): €75.00

=== Multi-Currency Shopping Carts ===

Shopping Cart (USD):
==================================================
Laptop                         $999.99
Mouse                    2 × $29.99 = $59.98
Keyboard                       $79.99
--------------------------------------------------
Subtotal                       $1,139.96
Tax (8%)                       $91.20
==================================================
TOTAL                          $1,231.16

Shopping Cart (EUR):
==================================================
Laptop                         €849.99
Mouse                    2 × €24.99 = €49.98
Keyboard                       €69.99
--------------------------------------------------
Subtotal                       €969.96
Tax (8%)                       €77.60
==================================================
TOTAL                          €1,047.56

Shopping Cart (GBP):
==================================================
```

```
Laptop                          £749.99
Mouse                     2 × £19.99 = £39.98
Keyboard                        £59.99
--------------------------------------------------
Subtotal                        £849.96
Tax (8%)                        £68.00
==================================================
TOTAL                           £917.96

=== User Currency Preferences ===
Alice's balance: €1000.00
Bob's balance: £1000.00
Charlie's balance: ¥1000
Dave's balance: €1000.00
```

10.12 Money and Inheritance

Using inheritance to extend the Money class for specialized financial applications.

Specialized Money Classes

```
from abc import ABC, abstractmethod
from decimal import Decimal
import datetime

class PaymentMethod(ABC):
    """Abstract base class for payment methods"""

    @abstractmethod
    def process_payment(self, amount: 'Money') -&gt; bool:
        """Process a payment"""
        pass

    @abstractmethod
    def get_fees(self, amount: 'Money') -&gt; 'Money':
        """Calculate processing fees"""
        pass

class CreditCardPayment(PaymentMethod):
    """Credit card payment processing"""

    def __init__(self, fee_rate: Decimal = Decimal('0.029')):  # 2.9% fee
        self.fee_rate = fee_rate

    def process_payment(self, amount: 'Money') -&gt; bool:
        """Process credit card payment"""
        print(f"Processing credit card payment of {amount}")
        return True

    def get_fees(self, amount: 'Money') -&gt; 'Money':
        """Calculate credit card processing fee"""
        fee_amount = amount * self.fee_rate
        return AdvancedMoney(fee_amount.amount, amount.currency)
```

```
class BankTransferPayment(PaymentMethod):
    """Bank transfer payment processing"""

    def __init__(self, flat_fee: 'Money' = None):
        self.flat_fee = flat_fee or AdvancedMoney('2.50', 'USD')

    def process_payment(self, amount: 'Money') -&gt; bool:
        """Process bank transfer"""
        print(f"Processing bank transfer of {amount}")
        return True

    def get_fees(self, amount: 'Money') -&gt; 'Money':
        """Calculate bank transfer fee"""
        if amount.currency != self.flat_fee.currency:
            raise IncompatibleCurrencyError("Currency mismatch for fee calculation")
        return self.flat_fee

class AccountBalance(AdvancedMoney):
    """Money subclass for account balances with transaction history"""

    def __init__(self, amount, currency='USD', account_type='checking'):
        super().__init__(amount, currency)
        self.account_type = account_type
        self.transactions = []
        self.created_date = datetime.datetime.now()

    def deposit(self, amount: 'AdvancedMoney', description: str = "Deposit"):
        """Make a deposit"""
        if amount.currency != self.currency:
            raise IncompatibleCurrencyError(f"Cannot deposit {amount.currency} to {self.c

        new_balance = AccountBalance(
            self.amount + amount.amount,
            self.currency,
            self.account_type
        )
        new_balance.transactions = self.transactions.copy()
        new_balance.transactions.append({
            'type': 'deposit',
            'amount': amount,
            'description': description,
            'timestamp': datetime.datetime.now(),
            'balance_after': new_balance
        })
        new_balance.created_date = self.created_date
        return new_balance

    def withdraw(self, amount: 'AdvancedMoney', description: str = "Withdrawal"):
        """Make a withdrawal"""
        if amount.currency != self.currency:
            raise IncompatibleCurrencyError(f"Cannot withdraw {amount.currency} from {sel

        if amount &gt; AdvancedMoney(self.amount, self.currency):
            raise ValueError("Insufficient funds")
```

```
        new_balance = AccountBalance(
            self.amount - amount.amount,
            self.currency,
            self.account_type
        )
        new_balance.transactions = self.transactions.copy()
        new_balance.transactions.append({
            'type': 'withdrawal',
            'amount': amount,
            'description': description,
            'timestamp': datetime.datetime.now(),
            'balance_after': new_balance
        })
        new_balance.created_date = self.created_date
        return new_balance

    def transfer_to(self, target_account: 'AccountBalance', amount: 'AdvancedMoney',
                    description: str = "Transfer"):
        """Transfer money to another account"""
        if amount.currency != self.currency or amount.currency != target_account.currency
            raise IncompatibleCurrencyError("Currency mismatch for transfer")

        # Withdraw from source
        new_source_balance = self.withdraw(amount, f"Transfer to {target_account.account_

        # Deposit to target
        new_target_balance = target_account.deposit(amount, f"Transfer from {self.account

        return new_source_balance, new_target_balance

    def get_transaction_history(self, limit: int = 10):
        """Get recent transaction history"""
        return self.transactions[-limit:]

    def calculate_interest(self, annual_rate: Decimal, days: int = 30):
        """Calculate interest for account (savings accounts)"""
        if self.account_type not in ['savings', 'money_market']:
            return AdvancedMoney('0', self.currency)

        daily_rate = annual_rate / Decimal('365')
        interest_amount = self.amount * daily_rate * days
        return AdvancedMoney(interest_amount, self.currency)

class InvestmentBalance(AccountBalance):
    """Specialized balance for investment accounts"""

    def __init__(self, amount, currency='USD'):
        super().__init__(amount, currency, 'investment')
        self.holdings = {}  # symbol -&gt; quantity
        self.cost_basis = {}  # symbol -&gt; average cost

    def buy_stock(self, symbol: str, quantity: int, price_per_share: 'AdvancedMoney'):
        """Buy stock"""
        total_cost = price_per_share * quantity

        if total_cost &gt; AdvancedMoney(self.amount, self.currency):
```

```
            raise ValueError("Insufficient funds for purchase")

        # Update balance
        new_balance = self.withdraw(total_cost, f"Buy {quantity} shares of {symbol}")

        # Update holdings
        if symbol in new_balance.holdings:
            old_quantity = new_balance.holdings[symbol]
            old_cost_basis = new_balance.cost_basis[symbol]

            # Calculate new average cost
            total_shares = old_quantity + quantity
            total_cost_basis = (old_quantity * old_cost_basis) + total_cost
            new_cost_basis = total_cost_basis / total_shares

            new_balance.holdings[symbol] = total_shares
            new_balance.cost_basis[symbol] = new_cost_basis
        else:
            new_balance.holdings[symbol] = quantity
            new_balance.cost_basis[symbol] = price_per_share

        return new_balance

    def sell_stock(self, symbol: str, quantity: int, price_per_share: 'AdvancedMoney'):
        """Sell stock"""
        if symbol not in self.holdings or self.holdings[symbol] &lt; quantity:
            raise ValueError(f"Insufficient shares of {symbol}")

        total_proceeds = price_per_share * quantity

        # Update balance
        new_balance = self.deposit(total_proceeds, f"Sell {quantity} shares of {symbol}")

        # Update holdings
        new_balance.holdings[symbol] -= quantity
        if new_balance.holdings[symbol] == 0:
            del new_balance.holdings[symbol]
            del new_balance.cost_basis[symbol]

        # Calculate gain/loss
        original_cost = self.cost_basis[symbol] * quantity
        gain_loss = total_proceeds - AdvancedMoney(original_cost.amount, self.currency)

        new_balance.transactions[-1]['gain_loss'] = gain_loss

        return new_balance

    def get_portfolio_value(self, current_prices: dict):
        """Calculate total portfolio value"""
        cash_value = AdvancedMoney(self.amount, self.currency)
        stock_value = AdvancedMoney('0', self.currency)

        for symbol, quantity in self.holdings.items():
            if symbol in current_prices:
                stock_value += current_prices[symbol] * quantity
```

```python
        return cash_value + stock_value

# Demonstrate inheritance hierarchy
print("=== Money Inheritance Hierarchy ===")

# Create accounts
checking = AccountBalance('1000.00', 'USD', 'checking')
savings = AccountBalance('5000.00', 'USD', 'savings')
investment = InvestmentBalance('10000.00', 'USD')

print(f"Initial balances:")
print(f"Checking: {checking}")
print(f"Savings: {savings}")
print(f"Investment: {investment}")

# Account operations
deposit_amount = AdvancedMoney('500.00', 'USD')
checking = checking.deposit(deposit_amount, "Payroll deposit")
print(f"\nAfter deposit: {checking}")

# Transfer between accounts
transfer_amount = AdvancedMoney('200.00', 'USD')
checking, savings = checking.transfer_to(savings, transfer_amount, "Emergency fund")
print(f"After transfer - Checking: {checking}, Savings: {savings}")

# Investment operations
apple_price = AdvancedMoney('150.00', 'USD')
investment = investment.buy_stock('AAPL', 10, apple_price)
print(f"After buying AAPL: {investment}")
print(f"Holdings: {investment.holdings}")

# Sell some stock
new_apple_price = AdvancedMoney('160.00', 'USD')
investment = investment.sell_stock('AAPL', 5, new_apple_price)
print(f"After selling 5 AAPL: {investment}")
print(f"Remaining holdings: {investment.holdings}")

# Check transaction history
print(f"\nRecent transactions:")
for i, transaction in enumerate(investment.get_transaction_history(3), 1):
    print(f"{i}. {transaction['type'].title()}: {transaction['amount']} - {transaction['c
    if 'gain_loss' in transaction:
        print(f"   Gain/Loss: {transaction['gain_loss']}")

# Payment processing demonstration
print(f"\n=== Payment Processing ===")

payment_amount = AdvancedMoney('100.00', 'USD')

# Credit card payment
cc_processor = CreditCardPayment()
cc_fee = cc_processor.get_fees(payment_amount)
print(f"Credit card payment of {payment_amount}")
print(f"Processing fee: {cc_fee}")
print(f"Total charge: {payment_amount + cc_fee}")
```

```
# Bank transfer payment
bt_processor = BankTransferPayment()
bt_fee = bt_processor.get_fees(payment_amount)
print(f"\nBank transfer payment of {payment_amount}")
print(f"Processing fee: {bt_fee}")
print(f"Total charge: {payment_amount + bt_fee}")

# Interest calculation
annual_rate = Decimal('0.02')  # 2% APY
interest = savings.calculate_interest(annual_rate, 30)
print(f"\nSavings account interest (2% APY, 30 days): {interest}")

# Portfolio valuation
current_prices = {'AAPL': AdvancedMoney('165.00', 'USD')}
portfolio_value = investment.get_portfolio_value(current_prices)
print(f"Total portfolio value: {portfolio_value}")
```

Output:

```
=== Money Inheritance Hierarchy ===
Initial balances:
Checking: $1,000.00
Savings: $5,000.00
Investment: $10,000.00

After deposit: $1,500.00

After transfer - Checking: $1,300.00, Savings: $5,200.00
After buying AAPL: $8,500.00
Holdings: {'AAPL': 10}
After selling 5 AAPL: $9,300.00
Remaining holdings: {'AAPL': 5}

Recent transactions:
1. Withdrawal: $1,500.00 - Buy 10 shares of AAPL
2. Deposit: $800.00 - Sell 5 shares of AAPL
   Gain/Loss: $50.00

=== Payment Processing ===
Credit card payment of $100.00
Processing fee: $2.90
Total charge: $102.90

Bank transfer payment of $100.00
Processing fee: $2.50
Total charge: $102.50

Savings account interest (2% APY, 30 days): $8.55

Total portfolio value: $10,125.00
```

10.13 The Fraction Class

The Fraction class provides exact rational number arithmetic, perfect for applications requiring precise fractional calculations.

Working with Fractions

```
from fractions import Fraction
import math

print("=== Fraction Class Basics ===")

# Creating fractions
f1 = Fraction(3, 4)        # 3/4
f2 = Fraction(1, 2)        # 1/2
f3 = Fraction('0.75')      # From decimal string
f4 = Fraction(0.5)         # From float (be careful!)
f5 = Fraction('1/3')       # From fraction string

print(f"f1 = {f1} = {f1.numerator}/{f1.denominator}")
print(f"f2 = {f2} = {f2.numerator}/{f2.denominator}")
print(f"f3 = {f3}")
print(f"f4 = {f4}")
print(f"f5 = {f5}")

# Automatic simplification
f6 = Fraction(6, 8)        # Automatically reduced to 3/4
print(f"Fraction(6, 8) = {f6}")

# Arithmetic operations
print(f"\n=== Fraction Arithmetic ===")
print(f"{f1} + {f2} = {f1 + f2}")
print(f"{f1} - {f2} = {f1 - f2}")
print(f"{f1} * {f2} = {f1 * f2}")
print(f"{f1} / {f2} = {f1 / f2}")

# Power operations
print(f"{f1}^2 = {f1 ** 2}")
print(f"{f2}^-1 = {f2 ** -1}")

# Comparison operations
print(f"\n=== Fraction Comparisons ===")
print(f"{f1} == {f3}: {f1 == f3}")
print(f"{f1} &gt; {f2}: {f1 &gt; f2}")
print(f"{f2} &lt; {f1}: {f2 &lt; f1}")

# Converting to other types
print(f"\n=== Type Conversions ===")
print(f"{f1} as float: {float(f1)}")
print(f"{f1} as decimal: {f1.numerator / f1.denominator}")

# Fraction methods
print(f"\n=== Fraction Methods ===")
mixed_fraction = Fraction(22, 7)  # Improper fraction
print(f"22/7 = {mixed_fraction} = {float(mixed_fraction):.6f}")
```

```python
print(f"limit_denominator(100): {Fraction(22/7).limit_denominator(100)}")
print(f"limit_denominator(10): {Fraction(22/7).limit_denominator(10)}")

# Working with recipes - practical application
class Recipe:
    """Recipe class using fractions for precise measurements"""

    def __init__(self, name: str):
        self.name = name
        self.ingredients = {}
        self.servings = 1

    def add_ingredient(self, name: str, amount: Fraction, unit: str):
        """Add ingredient with fractional amount"""
        self.ingredients[name] = {'amount': amount, 'unit': unit}

    def scale_recipe(self, new_servings: int):
        """Scale recipe to different number of servings"""
        scaling_factor = Fraction(new_servings, self.servings)
        scaled_recipe = Recipe(f"{self.name} (scaled for {new_servings})")

        for ingredient, details in self.ingredients.items():
            new_amount = details['amount'] * scaling_factor
            scaled_recipe.add_ingredient(ingredient, new_amount, details['unit'])

        scaled_recipe.servings = new_servings
        return scaled_recipe

    def __str__(self):
        result = f"{self.name} (serves {self.servings}):\n"
        for ingredient, details in self.ingredients.items():
            amount = details['amount']
            unit = details['unit']

            # Convert to mixed number for display if &gt; 1
            if amount &gt; 1:
                whole_part = amount.numerator // amount.denominator
                fractional_part = Fraction(amount.numerator % amount.denominator,
                                         amount.denominator)
                if fractional_part == 0:
                    amount_str = str(whole_part)
                else:
                    amount_str = f"{whole_part} {fractional_part}"
            else:
                amount_str = str(amount)

            result += f"  {amount_str} {unit} {ingredient}\n"

        return result.strip()

# Create a recipe
print(f"\n=== Recipe Application ===")

chocolate_chip_cookies = Recipe("Chocolate Chip Cookies")
chocolate_chip_cookies.add_ingredient("flour", Fraction(2, 1), "cups")
chocolate_chip_cookies.add_ingredient("sugar", Fraction(3, 4), "cup")
```

```
chocolate_chip_cookies.add_ingredient("butter", Fraction(1, 2), "cup")
chocolate_chip_cookies.add_ingredient("eggs", Fraction(1, 1), "large")
chocolate_chip_cookies.add_ingredient("vanilla", Fraction(1, 2), "teaspoon")
chocolate_chip_cookies.add_ingredient("baking soda", Fraction(1, 2), "teaspoon")
chocolate_chip_cookies.add_ingredient("chocolate chips", Fraction(1, 1), "cup")

print("Original recipe:")
print(chocolate_chip_cookies)

# Scale for different servings
scaled_recipe = chocolate_chip_cookies.scale_recipe(3)
print(f"\nScaled recipe:")
print(scaled_recipe)

# Half recipe
half_recipe = chocolate_chip_cookies.scale_recipe(1).scale_recipe(1/2)
print(f"\nHalf recipe:")
print(half_recipe)

# Fraction calculations for cooking
print(f"\n=== Cooking Calculations ===")

# Pizza cutting
pizza_slices = Fraction(1, 8)  # Each slice is 1/8 of pizza
people_eating = 3
slices_per_person = 2

total_pizza_needed = pizza_slices * slices_per_person * people_eating
print(f"Pizza needed: {people_eating} people × {slices_per_person} slices × {pizza_slices

# Fabric calculations
fabric_per_yard = Fraction(7, 8)  # 7/8 yard needed per item
items_to_make = 5
total_fabric = fabric_per_yard * items_to_make

print(f"Fabric needed: {items_to_make} items × {fabric_per_yard} yards = {total_fabric} y

# Convert to mixed number
if total_fabric &gt; 1:
    whole_yards = total_fabric.numerator // total_fabric.denominator
    remaining_fraction = Fraction(total_fabric.numerator % total_fabric.denominator,
                                  total_fabric.denominator)
    print(f"That's {whole_yards} and {remaining_fraction} yards")

# Mathematical applications
print(f"\n=== Mathematical Applications ===")

# Geometric series sum: 1 + 1/2 + 1/4 + 1/8 + ...
def geometric_series_sum(first_term: Fraction, ratio: Fraction, num_terms: int):
    """Calculate sum of geometric series using fractions"""
    total = Fraction(0)
    current_term = first_term

    for i in range(num_terms):
        total += current_term
        current_term *= ratio
```

```
    return total

# Sum of 1 + 1/2 + 1/4 + 1/8 + 1/16 (5 terms)
series_sum = geometric_series_sum(Fraction(1), Fraction(1, 2), 5)
print(f"Geometric series (5 terms): {series_sum} = {float(series_sum)}")

# Continued fractions
def continued_fraction_approximation(decimal_value: float, max_denominator: int = 1000):
    """Find continued fraction approximation"""
    return Fraction(decimal_value).limit_denominator(max_denominator)

# Approximate π
pi_approx = continued_fraction_approximation(math.pi, 100)
print(f"π ≈ {pi_approx} = {float(pi_approx):.6f}")
print(f"Error: {abs(math.pi - float(pi_approx)):.6f}")

# Approximate e
e_approx = continued_fraction_approximation(math.e, 100)
print(f"e ≈ {e_approx} = {float(e_approx):.6f}")
print(f"Error: {abs(math.e - float(e_approx)):.6f}")

# Egyptian fractions (sum of unit fractions)
def egyptian_fractions(frac: Fraction):
    """Convert fraction to sum of unit fractions (greedy algorithm)"""
    if frac == 0:
        return []

    result = []
    while frac &gt; 0:
        # Find smallest unit fraction &lt;= frac
        unit_denominator = -(-frac.denominator // frac.numerator)  # Ceiling division
        unit_fraction = Fraction(1, unit_denominator)

        result.append(unit_fraction)
        frac -= unit_fraction

    return result

# Convert 5/6 to Egyptian fractions
original_frac = Fraction(5, 6)
egyptian = egyptian_fractions(original_frac)
print(f"\n{original_frac} as Egyptian fractions: {' + '.join(str(f) for f in egyptian)}")
print(f"Verification: {sum(egyptian)} = {original_frac}")
```

Output:

```
=== Fraction Class Basics ===
f1 = 3/4 = 3/4
f2 = 1/2 = 1/2
f3 = 3/4
f4 = 1/2
f5 = 1/3

Fraction(6, 8) = 3/4
```

```
=== Fraction Arithmetic ===
3/4 + 1/2 = 5/4
3/4 - 1/2 = 1/4
3/4 * 1/2 = 3/8
3/4 / 1/2 = 3/2

3/4^2 = 9/16
1/2^-1 = 2

=== Fraction Comparisons ===
3/4 == 3/4: True
3/4 &gt; 1/2: True
1/2 &lt; 3/4: True

=== Type Conversions ===
3/4 as float: 0.75
3/4 as decimal: 0.75

=== Fraction Methods ===
22/7 = 22/7 = 3.142857
limit_denominator(100): 22/7
limit_denominator(10): 3/1

=== Recipe Application ===
Original recipe:
Chocolate Chip Cookies (serves 1):
  2 cups flour
  3/4 cup sugar
  1/2 cup butter
  1 large eggs
  1/2 teaspoon vanilla
  1/2 teaspoon baking soda
  1 cup chocolate chips

Scaled recipe:
Chocolate Chip Cookies (scaled for 3) (serves 3):
  6 cups flour
  2 1/4 cup sugar
  1 1/2 cup butter
  3 large eggs
  1 1/2 teaspoon vanilla
  1 1/2 teaspoon baking soda
  3 cup chocolate chips

Half recipe:
Chocolate Chip Cookies (serves 1) (serves 1):
  1 cups flour
  3/8 cup sugar
  1/4 cup butter
  1/2 large eggs
  1/4 teaspoon vanilla
  1/4 teaspoon baking soda
  1/2 cup chocolate chips

=== Cooking Calculations ===
```

```
Pizza needed: 3 people × 2 slices × 1/8 = 3/4 pizzas
Fabric needed: 5 items × 7/8 yards = 35/8 yards
That's 4 and 3/8 yards

=== Mathematical Applications ===
Geometric series (5 terms): 31/16 = 1.9375
π ≈ 22/7 = 3.142857
Error: 0.001265
e ≈ 19/7 = 2.714286
Error: 0.004032

5/6 as Egyptian fractions: 1/2 + 1/3
Verification: 5/6 = 5/6
```

10.14 The Complex Class

Python's built-in complex number support provides powerful tools for mathematical and engineering applications.

Complex Number Operations

```
import cmath
import math
import matplotlib.pyplot as plt
import numpy as np

print("=== Complex Number Basics ===")

# Creating complex numbers
c1 = complex(3, 4)         # 3 + 4j
c2 = 5 + 2j               # Direct notation
c3 = complex('2+3j')       # From string
c4 = complex(1)           # Real only: 1 + 0j

print(f"c1 = {c1}")
print(f"c2 = {c2}")
print(f"c3 = {c3}")
print(f"c4 = {c4}")

# Accessing real and imaginary parts
print(f"\nComplex number components:")
print(f"c1.real = {c1.real}")
print(f"c1.imag = {c1.imag}")

# Basic arithmetic operations
print(f"\n=== Complex Arithmetic ===")
print(f"c1 + c2 = {c1 + c2}")
print(f"c1 - c2 = {c1 - c2}")
print(f"c1 * c2 = {c1 * c2}")
print(f"c1 / c2 = {c1 / c2}")
print(f"c1 ** 2 = {c1 ** 2}")

# Complex conjugate
```

```python
print(f"\nconjugate of {c1} = {c1.conjugate()}")

# Absolute value (magnitude)
print(f"abs({c1}) = {abs(c1)}")

# Using cmath module for advanced operations
print(f"\n=== Advanced Complex Math ===")

# Polar form conversion
magnitude = abs(c1)
phase = cmath.phase(c1)
print(f"{c1} in polar form:")
print(f"  Magnitude: {magnitude}")
print(f"  Phase: {phase} radians = {math.degrees(phase):.2f} degrees")

# Convert back from polar
c1_from_polar = cmath.rect(magnitude, phase)
print(f"  Back to rectangular: {c1_from_polar}")

# Exponential form: z = re^(iθ)
print(f"\nExponential operations:")
exp_result = cmath.exp(c1)
print(f"e^{c1} = {exp_result}")

log_result = cmath.log(c1)
print(f"ln({c1}) = {log_result}")

sqrt_result = cmath.sqrt(c1)
print(f"√{c1} = {sqrt_result}")

# Trigonometric functions with complex numbers
print(f"\nTrigonometric functions:")
print(f"sin({c1}) = {cmath.sin(c1)}")
print(f"cos({c1}) = {cmath.cos(c1)}")
print(f"tan({c1}) = {cmath.tan(c1)}")

# Practical applications
class ComplexSignalProcessor:
    """Complex number applications in signal processing"""

    def __init__(self):
        self.sample_rate = 1000  # Hz

    def generate_complex_sinusoid(self, frequency: float, duration: float, amplitude: flo
        """Generate complex sinusoid: A * e^(2πift)"""
        t = np.linspace(0, duration, int(self.sample_rate * duration))
        signal = amplitude * np.exp(2j * np.pi * frequency * t)
        return t, signal

    def fourier_transform_sample(self, signal):
        """Simple DFT example"""
        return np.fft.fft(signal)

    def impedance_calculation(self, resistance: float, reactance: float):
        """Calculate electrical impedance Z = R + jX"""
        impedance = complex(resistance, reactance)
```

```
        magnitude = abs(impedance)
        phase_deg = math.degrees(cmath.phase(impedance))

        return {
            'impedance': impedance,
            'magnitude': magnitude,
            'phase_degrees': phase_deg,
            'phase_radians': cmath.phase(impedance)
        }

# Signal processing example
print(f"\n=== Signal Processing Application ===")

processor = ComplexSignalProcessor()

# Generate a complex sinusoid
frequency = 50  # Hz
duration = 0.1  # seconds
t, signal = processor.generate_complex_sinusoid(frequency, duration)

print(f"Generated complex sinusoid:")
print(f"Frequency: {frequency} Hz")
print(f"Duration: {duration} seconds")
print(f"Sample points: {len(signal)}")
print(f"First few samples:")
for i in range(5):
    print(f"  t={t[i]:.3f}s: {signal[i]:.3f}")

# Electrical impedance example
print(f"\n=== Electrical Impedance Calculation ===")

# AC circuit analysis
resistor = 100  # Ohms
inductor_reactance = 50  # Ohms (ωL)
capacitor_reactance = -75  # Ohms (-1/ωC)

# Series RLC circuit
total_reactance = inductor_reactance + capacitor_reactance
impedance_data = processor.impedance_calculation(resistor, total_reactance)

print(f"Circuit components:")
print(f"  Resistance: {resistor} Ω")
print(f"  Inductive reactance: {inductor_reactance} Ω")
print(f"  Capacitive reactance: {capacitor_reactance} Ω")
print(f"  Total reactance: {total_reactance} Ω")

print(f"\nTotal impedance: {impedance_data['impedance']:.2f}")
print(f"Magnitude: {impedance_data['magnitude']:.2f} Ω")
print(f"Phase: {impedance_data['phase_degrees']:.2f}°")

# Mandelbrot set calculation
class MandelbrotSet:
    """Generate points in the Mandelbrot set"""

    def __init__(self, max_iterations=100):
        self.max_iterations = max_iterations
```

```
    def mandelbrot_point(self, c: complex) -&gt; int:
        """Calculate iterations for a point in the Mandelbrot set"""
        z = 0
        for n in range(self.max_iterations):
            if abs(z) &gt; 2:
                return n
            z = z * z + c
        return self.max_iterations

    def generate_mandelbrot_data(self, x_range, y_range, resolution=100):
        """Generate Mandelbrot set data"""
        x_min, x_max = x_range
        y_min, y_max = y_range

        x = np.linspace(x_min, x_max, resolution)
        y = np.linspace(y_min, y_max, resolution)

        mandelbrot_data = np.zeros((resolution, resolution))

        for i, real in enumerate(x):
            for j, imag in enumerate(y):
                c = complex(real, imag)
                mandelbrot_data[j, i] = self.mandelbrot_point(c)

        return x, y, mandelbrot_data

# Mandelbrot example
print(f"\n=== Mandelbrot Set Example ===")

mandelbrot = MandelbrotSet(max_iterations=50)

# Test some specific points
test_points = [
    complex(0, 0),      # In the set
    complex(-1, 0),     # In the set
    complex(-0.5, 0.5), # In the set
    complex(1, 1),      # Not in the set
    complex(0.5, 0.5)   # Not in the set
]

print("Testing points in Mandelbrot set:")
for point in test_points:
    iterations = mandelbrot.mandelbrot_point(point)
    if iterations == mandelbrot.max_iterations:
        print(f"  {point}: In the set (converged)")
    else:
        print(f"  {point}: Not in the set (diverged after {iterations} iterations)")

# Quadratic formula with complex solutions
def solve_quadratic(a, b, c):
    """Solve ax² + bx + c = 0 using complex arithmetic"""
    discriminant = b**2 - 4*a*c

    if discriminant &gt;= 0:
        # Real solutions
```

```
        sqrt_discriminant = math.sqrt(discriminant)
        x1 = (-b + sqrt_discriminant) / (2*a)
        x2 = (-b - sqrt_discriminant) / (2*a)
        return x1, x2
    else:
        # Complex solutions
        sqrt_discriminant = cmath.sqrt(discriminant)
        x1 = (-b + sqrt_discriminant) / (2*a)
        x2 = (-b - sqrt_discriminant) / (2*a)
        return x1, x2

print(f"\n=== Quadratic Equation Solver ===")

# Test cases
equations = [
    (1, -5, 6),     # x² - 5x + 6 = 0 (real solutions)
    (1, 0, 4),      # x² + 4 = 0 (pure imaginary)
    (1, -2, 5),     # x² - 2x + 5 = 0 (complex solutions)
    (2, -4, 5)      # 2x² - 4x + 5 = 0 (complex solutions)
]

for a, b, c in equations:
    print(f"\nSolving {a}x² + {b}x + {c} = 0:")
    x1, x2 = solve_quadratic(a, b, c)
    print(f"  x₁ = {x1}")
    print(f"  x₂ = {x2}")

    # Verify solutions
    def verify_solution(x):
        return a*x**2 + b*x + c

    print(f"  Verification:")
    print(f"    f(x₁) = {verify_solution(x1)}")
    print(f"    f(x₂) = {verify_solution(x2)}")

# Custom complex class for educational purposes
class Educational_Complex:
    """Educational complex number class to demonstrate implementation"""

    def __init__(self, real=0, imag=0):
        self.real = float(real)
        self.imag = float(imag)

    def __str__(self):
        if self.imag &gt;= 0:
            return f"{self.real}+{self.imag}j"
        else:
            return f"{self.real}{self.imag}j"

    def __repr__(self):
        return f"Educational_Complex({self.real}, {self.imag})"

    def __add__(self, other):
        if isinstance(other, Educational_Complex):
            return Educational_Complex(self.real + other.real, self.imag + other.imag)
        else:
```

```
            return Educational_Complex(self.real + other, self.imag)

    def __mul__(self, other):
        if isinstance(other, Educational_Complex):
            # (a + bi)(c + di) = (ac - bd) + (ad + bc)i
            real_part = self.real * other.real - self.imag * other.imag
            imag_part = self.real * other.imag + self.imag * other.real
            return Educational_Complex(real_part, imag_part)
        else:
            return Educational_Complex(self.real * other, self.imag * other)

    def conjugate(self):
        return Educational_Complex(self.real, -self.imag)

    def magnitude(self):
        return math.sqrt(self.real**2 + self.imag**2)

    def phase(self):
        return math.atan2(self.imag, self.real)

# Test educational complex class
print(f"\n=== Educational Complex Class ===")

edu_c1 = Educational_Complex(3, 4)
edu_c2 = Educational_Complex(1, -2)

print(f"edu_c1 = {edu_c1}")
print(f"edu_c2 = {edu_c2}")
print(f"edu_c1 + edu_c2 = {edu_c1 + edu_c2}")
print(f"edu_c1 * edu_c2 = {edu_c1 * edu_c2}")
print(f"conjugate of edu_c1 = {edu_c1.conjugate()}")
print(f"magnitude of edu_c1 = {edu_c1.magnitude():.3f}")
print(f"phase of edu_c1 = {edu_c1.phase():.3f} radians = {math.degrees(edu_c1.phase()):.1
```

Output:

```
=== Complex Number Basics ===
c1 = (3+4j)
c2 = (5+2j)
c3 = (2+3j)
c4 = (1+0j)

Complex number components:
c1.real = 3.0
c1.imag = 4.0

=== Complex Arithmetic ===
c1 + c2 = (8+6j)
c1 - c2 = (-2+2j)
c1 * c2 = (7+26j)
c1 / c2 = (0.7931034482758621+0.3103448275862069j)
c1 ** 2 = (-7+24j)

conjugate of (3+4j) = (3-4j)
abs((3+4j)) = 5.0
```

```
=== Advanced Complex Math ===
(3+4j) in polar form:
  Magnitude: 5.0
  Phase: 0.9272952180016122 radians = 53.13 degrees
  Back to rectangular: (3.0000000000000004+3.9999999999999996j)

Exponential operations:
e^(3+4j) = (-13.128783081462158+15.200784463067954j)
ln((3+4j)) = (1.6094379124341003+0.9272952180016122j)
√(3+4j) = (2.0+1.0j)

Trigonometric functions:
sin((3+4j)) = (3.853738037919377-27.016813258003932j)
cos((3+4j)) = (-27.034945603074224-3.851153334811777j)
tan((3+4j)) = (-0.00018734620462947045+0.9993559873814731j)

=== Signal Processing Application ===
Generated complex sinusoid:
Frequency: 50 Hz
Duration: 0.1 seconds
Sample points: 100
First few samples:
  t=0.000s: 1.000+0.000j
  t=0.001s: 0.988+0.156j
  t=0.002s: 0.951+0.309j
  t=0.003s: 0.891+0.454j
  t=0.004s: 0.809+0.588j

=== Electrical Impedance Calculation ===
Circuit components:
  Resistance: 100 Ω
  Inductive reactance: 50 Ω
  Capacitive reactance: -75 Ω
  Total reactance: -25 Ω

Total impedance: (100.00-25.00j)
Magnitude: 103.08 Ω
Phase: -14.04°

=== Mandelbrot Set Example ===
Testing points in Mandelbrot set:
  0j: In the set (converged)
  (-1+0j): In the set (converged)
  (-0.5+0.5j): In the set (converged)
  (1+1j): Not in the set (diverged after 1 iterations)
  (0.5+0.5j): Not in the set (diverged after 4 iterations)

=== Quadratic Equation Solver ===

Solving 1x² + -5x + 6 = 0:
  x₁ = 3.0
  x₂ = 2.0
  Verification:
    f(x₁) = 0.0
    f(x₂) = 0.0
```

```
Solving 1x² + 0x + 4 = 0:
  x₁ = 2j
  x₂ = (-0-2j)
  Verification:
    f(x₁) = 0j
    f(x₂) = 0j

Solving 1x² + -2x + 5 = 0:
  x₁ = (1+2j)
  x₂ = (1-2j)
  Verification:
    f(x₁) = 0j
    f(x₂) = 0j

Solving 2x² + -4x + 5 = 0:
  x₁ = (1+1j)
  x₂ = (1-1j)
  Verification:
    f(x₁) = 0j
    f(x₂) = 0j

=== Educational Complex Class ===
edu_c1 = 3.0+4.0j
edu_c2 = 1.0+-2.0j
edu_c1 + edu_c2 = 4.0+2.0j
edu_c1 * edu_c2 = 11.0+-2.0j
conjugate of edu_c1 = 3.0+-4.0j
magnitude of edu_c1 = 5.000
phase of edu_c1 = 0.927 radians = 53.1°
```

Summary

This comprehensive tutorial has covered all aspects of Python's advanced numeric classes from Chapter 10:

Key Takeaways

1. **Decimal Class**: Provides exact decimal arithmetic, essential for financial applications where precision matters.
2. **Money Classes**: Custom implementation using containment and inheritance to handle monetary calculations with currency awareness.
3. **Fraction Class**: Enables exact rational number arithmetic, perfect for recipes, measurements, and mathematical applications.
4. **Complex Class**: Built-in support for complex numbers with applications in engineering, signal processing, and mathematics.

Best Practices

- Use `Decimal` for financial calculations requiring precision
- Implement `Money` classes for currency-aware applications
- Apply `Fraction` for exact fractional arithmetic
- Leverage `complex` numbers for mathematical and engineering calculations
- Always validate currency codes and handle edge cases
- Implement proper string representations (`__str__` and `__repr__`)
- Use context managers for temporary settings changes

Real-World Applications

- Financial systems and accounting software
- Recipe scaling and cooking applications
- Engineering calculations with complex numbers
- Scientific computing requiring exact arithmetic
- International commerce with multiple currencies

This tutorial provides a solid foundation for working with Python's advanced numeric classes, combining theoretical understanding with practical implementations that can be adapted for real-world applications.
[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43]

⁂

1. https://pypi.org/project/money/
2. https://www.scaler.com/topics/python-decimal/
3. https://www.geeksforgeeks.org/python/precision-handling-python/
4. https://www.geeksforgeeks.org/python/python-program-to-create-bankaccount-class-with-deposit-withdraw-function/
5. https://docs.python.org/3/library/decimal.html
6. https://www.linkedin.com/pulse/navigating-nuances-numeric-precision-deep-dive-python-utkarsh-singh-ccemc
7. https://github.com/vimeo/py-money
8. https://www.w3schools.com/Python/ref_module_decimal.asp
9. https://testdriven.io/tips/c1f6f393-5fa2-4edd-8b64-cc1344219173/
10. https://pypi.org/project/py-money/
11. CHAPTER-10.pdf
12. https://www.geeksforgeeks.org/python/decimal-functions-python-set-1/
13. https://www.reddit.com/r/learnpython/comments/ly5jv4/whats_different_between_decimal_and_floating/
14. https://github.com/carlospalol/money
15. https://www.geeksforgeeks.org/python/setting-precision-in-python-using-decimal-module/

16. https://dev.to/kalebu/when-decimal-is-better-than-float-in-python-io6
17. http://pythonfiddle.com/simplistic-money-class/
18. https://www.digitalocean.com/community/tutorials/python-decimal-division-round-precision
19. https://stackoverflow.com/questions/41453307/decimal-python-vs-float-runtime
20. https://profound.academy/python-mid/fraction-class-xxyi3ExuVKFf7o8QdEjL
21. https://zetcode.com/python/complex-builtin/
22. CHAPTER-10.pdf
23. https://www.tutorialspoint.com/fraction-module-in-python
24. https://www.geeksforgeeks.org/python/python-program-for-addition-and-subtraction-of-complex-numbers/
25. https://www.reddit.com/r/learnpython/comments/r39zcg/how_can_i_improve_this_fraction_class/
26. https://www.youtube.com/watch?v=AYsyz738BBk
27. https://www.geeksforgeeks.org/python/fraction-module-python/
28. https://www.w3schools.com/python/ref_func_complex.asp
29. https://python.plainenglish.io/learn-python-classes-by-building-a-banking-system-ab3205e8036b
30. https://runestone.academy/ns/books/published/pythonds/Introduction/ObjectOrientedProgramminginPythonDefiningClasses.html
31. https://www.python-engineer.com/posts/complex-numbers-python/
32. https://www.youtube.com/watch?v=8aW3tkIul-8
33. CHAPTER-10.pdf
34. https://realpython.com/python-fractions/
35. https://www.geeksforgeeks.org/python/complex-numbers-in-python-set-1-introduction/
36. https://github.com/py-moneyed/py-moneyed
37. https://www.youtube.com/watch?v=-3tDFL6Ch-s
38. http://hplgit.github.io/primer.html/doc/pub/class/._class-readable005.html
39. CHAPTER-10.pdf
40. CHAPTER-10.pdf
41. CHAPTER-10.pdf
42. https://www.pythonforbeginners.com/basics/decimal-module-in-python
43. https://www.laac.dev/blog/float-vs-decimal-python/