Complete Guide to Serialization and Deserialization in Python

Part 1: Understanding Serialization - The Magic of Data Transformation

What is Serialization?

Serialization is like taking a complex LEGO castle and breaking it down into individual pieces that fit in a box for shipping. **Deserialization** is rebuilding that castle from the pieces.

In programming terms:

- **Serialization**: Converting Python objects → Storable/Transmittable format
- **Deserialization**: Converting stored format → Python objects

Real-World Analogy

Think of serialization like **packing for a trip**:

```python
# Your Python object is like your stuff at home
my_data = {
    'clothes': ['shirt', 'pants', 'socks'],
    'electronics': {'phone': 'iPhone', 'laptop': 'MacBook'},
    'documents': ['passport', 'tickets']
}

# Serialization = Packing everything in a suitcase
# Deserialization = Unpacking when you arrive
```

Part 2: JSON Serialization - The Universal Language

Why JSON?

JSON (JavaScript Object Notation) is like the "English" of data formats - almost everything understands it!

Basic JSON Operations

```python
```

```python
import json

# Sample data - a social media profile
user_profile = {
    'username': 'alice_codes',
    'age': 28,
    'is_verified': True,
    'followers_count': 1250,
    'following': ['bob_dev', 'charlie_ai', 'diana_data'],
    'profile_settings': {
        'privacy': 'public',
        'notifications': True,
        'theme': 'dark'
    },
    'last_login': None
}

print("Original data:")
print(user_profile)
print(f"Type: {type(user_profile)}")

# SERIALIZATION - Python object to JSON string
json_string = json.dumps(user_profile, indent=2)  # indent makes it pretty
print("\nSerialized to JSON:")
print(json_string)
print(f"Type: {type(json_string)}")

# DESERIALIZATION - JSON string back to Python object
restored_profile = json.loads(json_string)
print("\nDeserialized back to Python:")
print(restored_profile)
print(f"Type: {type(restored_profile)}")

# Verify they're the same
print(f"\nData preserved correctly: {user_profile == restored_profile}")
```

JSON File Operations

python

```python
import json
import os

class ProfileManager:
    """Manages user profiles with JSON serialization"""

    def __init__(self, storage_dir="profiles"):
        self.storage_dir = storage_dir
        # Create directory if it doesn't exist
        os.makedirs(storage_dir, exist_ok=True)

    def save_profile(self, username, profile_data):
        """Save a user profile to JSON file"""
        filename = f"{self.storage_dir}/{username}.json"

        try:
            with open(filename, 'w', encoding='utf-8') as file:
                json.dump(profile_data, file, indent=2, ensure_ascii=False)
            return f"Profile saved successfully to {filename}"

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

    def load_profile(self, username):
        """Load a user profile from JSON file"""
        filename = f"{self.storage_dir}/{username}.json"

        try:
            with open(filename, 'r', encoding='utf-8') as file:
                profile_data = json.load(file)
            return profile_data

        except FileNotFoundError:
            return f"Profile for {username} not found"
        except json.JSONDecodeError as e:
            return f"Error reading JSON file: {e}"
        except Exception as e:
            return f"Unexpected error: {e}"

    def update_profile(self, username, updates):
        """Update specific fields in a user profile"""
        current_profile = self.load_profile(username)
```

```python
        if isinstance(current_profile, dict):
            # Merge updates with existing profile
            current_profile.update(updates)
            return self.save_profile(username, current_profile)
        else:
            return current_profile  # Return error message

    def list_profiles(self):
        """List all saved profiles"""
        try:
            files = [f for f in os.listdir(self.storage_dir) if f.endswith('.json')]
            usernames = [f.replace('.json', '') for f in files]
            return usernames
        except Exception as e:
            return f"Error listing profiles: {e}"

# Usage example
manager = ProfileManager()

# Create sample profiles
profiles = {
    'alice_codes': {
        'full_name': 'Alice Johnson',
        'age': 28,
        'skills': ['Python', 'JavaScript', 'React'],
        'experience_years': 5,
        'location': 'San Francisco',
        'is_looking_for_job': False
    },
    'bob_dev': {
        'full_name': 'Bob Smith',
        'age': 32,
        'skills': ['Java', 'Spring', 'Docker'],
        'experience_years': 8,
        'location': 'New York',
        'is_looking_for_job': True
    }
}

# Save profiles
for username, profile in profiles.items():
    result = manager.save_profile(username, profile)
    print(result)
```

```python
# Load and display profiles
print("\nStored profiles:")
for username in manager.list_profiles():
    profile = manager.load_profile(username)
    print(f"\n{username}:")
    print(f"  Name: {profile['full_name']}")
    print(f"  Skills: {', '.join(profile['skills'])}")
    print(f"  Looking for job: {profile['is_looking_for_job']}")

# Update a profile
update_result = manager.update_profile('alice_codes', {
    'skills': ['Python', 'JavaScript', 'React', 'TypeScript'],
    'is_looking_for_job': True
})
print(f"\nUpdate result: {update_result}")
```

Advanced JSON Serialization

python

```
import json
from datetime import datetime, date
from decimal import Decimal

class AdvancedJSONEncoder(json.JSONEncoder):
    """Custom JSON encoder for complex data types"""

    def default(self, obj):
        if isinstance(obj, datetime):
            return {
                '__type__': 'datetime',
                '__value__': obj.isoformat()
            }
        elif isinstance(obj, date):
            return {
                '__type__': 'date',
                '__value__': obj.isoformat()
            }
        elif isinstance(obj, Decimal):
            return {
                '__type__': 'decimal',
                '__value__': str(obj)
            }
        elif isinstance(obj, set):
            return {
                '__type__': 'set',
                '__value__': list(obj)
            }
        # Handle custom objects
        elif hasattr(obj, '__dict__'):
            return {
                '__type__': 'object',
                '__class__': obj.__class__.__name__,
                '__value__': obj.__dict__
            }

        return super().default(obj)

def advanced_json_decoder(dct):
    """Custom JSON decoder for complex data types"""
    if '__type__' in dct:
        obj_type = dct['__type__']
        value = dct['__value__']
```

```
            if obj_type == 'datetime':
                return datetime.fromisoformat(value)
            elif obj_type == 'date':
                return date.fromisoformat(value)
            elif obj_type == 'decimal':
                return Decimal(value)
            elif obj_type == 'set':
                return set(value)
            elif obj_type == 'object':
                # For demonstration - in real code, you'd properly reconstruct the object
                return {'_reconstructed_object': value, '_class': dct['__class__']}

        return dct

# Example with complex data types
class Transaction:
    def __init__(self, amount, description, timestamp=None):
        self.amount = Decimal(str(amount))
        self.description = description
        self.timestamp = timestamp or datetime.now()
        self.tags = set()

    def add_tag(self, tag):
        self.tags.add(tag)

# Create complex data
transaction = Transaction(99.99, "Coffee shop purchase")
transaction.add_tag("food")
transaction.add_tag("daily")

complex_data = {
    'transaction': transaction,
    'processed_date': date.today(),
    'processing_time': datetime.now(),
    'precision_amount': Decimal('123.456789'),
    'categories': {'food', 'beverage', 'daily'}
}

print("Original complex data:")
print(f"Transaction amount: {complex_data['transaction'].amount}")
print(f"Tags: {complex_data['transaction'].tags}")
print(f"Date: {complex_data['processed_date']}")
```

```
# Serialize with custom encoder
json_string = json.dumps(complex_data, cls=AdvancedJSONEncoder, indent=2)
print("\nSerialized JSON:")
print(json_string)

# Deserialize with custom decoder
restored_data = json.loads(json_string, object_hook=advanced_json_decoder)
print("\nRestored data:")
print(f"Transaction: {restored_data['transaction']}")
print(f"Date: {restored_data['processed_date']}")
print(f"Categories: {restored_data['categories']}")
```

Part 3: Pickle Serialization - Python's Native Format

Understanding Pickle

Pickle is like Python's own special language - it can save almost ANY Python object, but only Python can understand it.

```
python
```

```
import pickle
import datetime
from collections import defaultdict

class GameCharacter:
    """Example class for pickle demonstration"""

    def __init__(self, name, level=1):
        self.name = name
        self.level = level
        self.inventory = defaultdict(int)
        self.skills = {}
        self.last_played = datetime.datetime.now()

    def add_item(self, item, quantity=1):
        self.inventory[item] += quantity

    def learn_skill(self, skill, level):
        self.skills[skill] = level

    def level_up(self):
        self.level += 1
        self.last_played = datetime.datetime.now()

    def __str__(self):
        return f"Character: {self.name} (Level {self.level})"

    def get_save_summary(self):
        return {
            'name': self.name,
            'level': self.level,
            'item_count': sum(self.inventory.values()),
            'skill_count': len(self.skills),
            'last_played': self.last_played.strftime('%Y-%m-%d %H:%M:%S')
        }

class GameSaveManager:
    """Manages game saves using pickle"""

    def __init__(self, save_dir="game_saves"):
        self.save_dir = save_dir
        os.makedirs(save_dir, exist_ok=True)
```

```
def save_character(self, character, save_slot=1):
    """Save character to pickle file"""
    filename = f"{self.save_dir}/character_slot_{save_slot}.pkl"

    try:
        with open(filename, 'wb') as file:
            pickle.dump(character, file)
        return f"Character saved to slot {save_slot}"

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

def load_character(self, save_slot=1):
    """Load character from pickle file"""
    filename = f"{self.save_dir}/character_slot_{save_slot}.pkl"

    try:
        with open(filename, 'rb') as file:
            character = pickle.load(file)
        return character

    except FileNotFoundError:
        return f"No save found in slot {save_slot}"
    except Exception as e:
        return f"Error loading character: {e}"

def list_saves(self):
    """List all available save files with summaries"""
    saves = []
    try:
        files = [f for f in os.listdir(self.save_dir) if f.endswith('.pkl')]

        for filename in sorted(files):
            slot = filename.split('_')[-1].replace('.pkl', '')
            try:
                character = self.load_character(int(slot))
                if isinstance(character, GameCharacter):
                    saves.append({
                        'slot': slot,
                        'summary': character.get_save_summary()
                    })
            except:
                saves.append({
                    'slot': slot,
```

```
                'summary': 'Corrupted save file'
            })

        return saves

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

# Demonstration
save_manager = GameSaveManager()

# Create and develop a character
hero = GameCharacter("Sir Pythonicus")
hero.learn_skill("Coding", 95)
hero.learn_skill("Debugging", 87)
hero.learn_skill("Problem Solving", 92)

hero.add_item("Health Potion", 5)
hero.add_item("Magic Sword", 1)
hero.add_item("Gold Coins", 150)

for _ in range(10):  # Level up 10 times
    hero.level_up()

print("Original character:")
print(hero)
print(f"Inventory: {dict(hero.inventory)}")
print(f"Skills: {hero.skills}")

# Save the character
save_result = save_manager.save_character(hero, save_slot=1)
print(f"\nSave result: {save_result}")

# Load the character
loaded_hero = save_manager.load_character(save_slot=1)
print(f"\nLoaded character:")
print(loaded_hero)
print(f"Loaded inventory: {dict(loaded_hero.inventory)}")
print(f"Loaded skills: {loaded_hero.skills}")

# Verify the objects are equivalent but not the same
print(f"\nData preserved: {loaded_hero.name == hero.name and loaded_hero.level == hero.level}")
print(f"Same object: {loaded_hero is hero}")  # Should be False
```

```python
# List all saves
print("\nAvailable saves:")
for save_info in save_manager.list_saves():
    print(f"Slot {save_info['slot']}: {save_info['summary']}")
```

Pickle vs JSON Comparison

python

```
import pickle
import json
import sys
from datetime import datetime

class ComparisonDemo:
    """Demonstrate differences between pickle and JSON"""

    @staticmethod
    def compare_serialization_methods():
        # Test data with various Python types
        test_data = {
            'string': 'Hello World',
            'integer': 42,
            'float': 3.14159,
            'boolean': True,
            'none_value': None,
            'list': [1, 2, 3, 'four'],
            'dict': {'nested': 'value'},
            'tuple': (1, 2, 3),  # JSON doesn't preserve tuples
            'set': {1, 2, 3},    # JSON doesn't support sets
            'datetime': datetime.now(),  # JSON doesn't support datetime
        }

        print("=== SERIALIZATION COMPARISON ===\n")

        # JSON serialization (with limitations)
        try:
            # Create JSON-compatible version
            json_data = {
                'string': test_data['string'],
                'integer': test_data['integer'],
                'float': test_data['float'],
                'boolean': test_data['boolean'],
                'none_value': test_data['none_value'],
                'list': test_data['list'],
                'dict': test_data['dict'],
                'tuple_as_list': list(test_data['tuple']),  # Convert tuple to list
                'set_as_list': list(test_data['set']),      # Convert set to list
                'datetime_as_string': test_data['datetime'].isoformat()  # Convert datetime to string
            }

            json_string = json.dumps(json_data, indent=2)
```

```
    json_size = len(json_string.encode('utf-8'))

    print("JSON Serialization:")
    print(f"✓ Success - Size: {json_size} bytes")
    print("✓ Human-readable")
    print("✓ Language-independent")
    print("✗ Limited data types")
    print("✗ Data type information lost (tuple → list, set → list)")

except Exception as e:
    print(f"JSON Serialization failed: {e}")

print("\n" + "="*50 + "\n")

# Pickle serialization
try:
    pickle_bytes = pickle.dumps(test_data)
    pickle_size = len(pickle_bytes)

    print("Pickle Serialization:")
    print(f"✓ Success - Size: {pickle_size} bytes")
    print("✓ Preserves all Python data types")
    print("✓ Preserves object references")
    print("✗ Python-only format")
    print("✗ Not human-readable")
    print("✗ Security risks with untrusted data")

except Exception as e:
    print(f"Pickle Serialization failed: {e}")

# Deserialization comparison
print("\n" + "="*50 + "\n")
print("DESERIALIZATION RESULTS:")

# JSON deserialization
json_restored = json.loads(json_string)
print("JSON - Data type preservation:")
print(f"  Original tuple type: {type(test_data['tuple'])}")
print(f"  Restored as: {type(json_restored['tuple_as_list'])}")
print(f"  Original set type: {type(test_data['set'])}")
print(f"  Restored as: {type(json_restored['set_as_list'])}")

# Pickle deserialization
pickle_restored = pickle.loads(pickle_bytes)
```

```
    print("\nPickle - Data type preservation:")
    print(f"  Original tuple type: {type(test_data['tuple'])}")
    print(f"  Restored as: {type(pickle_restored['tuple'])}")
    print(f"  Original set type: {type(test_data['set'])}")
    print(f"  Restored as: {type(pickle_restored['set'])}")
    print(f"  Data completely preserved: {test_data == pickle_restored}")

# Run comparison
ComparisonDemo.compare_serialization_methods()
```

Part 4: Alternative Serialization Formats

YAML Serialization (Human-Friendly)

python

```
# Note: You need to install PyYAML: pip install PyYAML
try:
    import yaml

    class YAMLManager:
        """YAML serialization for configuration files"""

        @staticmethod
        def save_config(config_data, filename):
            """Save configuration to YAML file"""
            try:
                with open(filename, 'w') as file:
                    yaml.dump(config_data, file, default_flow_style=False, indent=2)
                return f"Configuration saved to {filename}"
            except Exception as e:
                return f"Error saving YAML: {e}"

        @staticmethod
        def load_config(filename):
            """Load configuration from YAML file"""
            try:
                with open(filename, 'r') as file:
                    return yaml.safe_load(file)
            except Exception as e:
                return f"Error loading YAML: {e}"

    # Example configuration
    app_config = {
        'database': {
            'host': 'localhost',
            'port': 5432,
            'name': 'myapp_db',
            'credentials': {
                'username': 'admin',
                'password': 'secret123'
            }
        },
        'api': {
            'version': '1.0',
            'rate_limits': {
                'requests_per_minute': 100,
                'burst_limit': 20
            },
```

```
            'endpoints': [
                '/api/users',
                '/api/posts',
                '/api/comments'
            ]
        },
        'features': {
            'enable_caching': True,
            'enable_logging': True,
            'debug_mode': False
        }
    }

    # Save and load YAML
    yaml_manager = YAMLManager()
    result = yaml_manager.save_config(app_config, 'app_config.yaml')
    print(f"YAML save result: {result}")

    loaded_config = yaml_manager.load_config('app_config.yaml')
    print("Loaded YAML config:")
    print(f"Database host: {loaded_config['database']['host']}")
    print(f"API version: {loaded_config['api']['version']}")

except ImportError:
    print("PyYAML not installed. Install with: pip install PyYAML")
```

XML Serialization (Enterprise-Friendly)

python

```
import xml.etree.ElementTree as ET
from xml.dom import minidom

class XMLManager:
    """XML serialization for structured data"""

    @staticmethod
    def dict_to_xml(data, root_name='root'):
        """Convert dictionary to XML"""

        def add_to_element(parent, key, value):
            """Recursively add data to XML element"""
            if isinstance(value, dict):
                child = ET.SubElement(parent, key)
                for k, v in value.items():
                    add_to_element(child, k, v)
            elif isinstance(value, list):
                for item in value:
                    if isinstance(item, dict):
                        child = ET.SubElement(parent, key)
                        for k, v in item.items():
                            add_to_element(child, k, v)
                    else:
                        child = ET.SubElement(parent, key)
                        child.text = str(item)
            else:
                child = ET.SubElement(parent, key)
                child.text = str(value)

        root = ET.Element(root_name)
        for key, value in data.items():
            add_to_element(root, key, value)

        return root

    @staticmethod
    def prettify_xml(element):
        """Return pretty-printed XML string"""
        rough_string = ET.tostring(element, 'unicode')
        reparsed = minidom.parseString(rough_string)
        return reparsed.toprettyxml(indent="  ")

    @staticmethod
```

```
    def save_xml(data, filename, root_name='data'):
        """Save dictionary as XML file"""
        try:
            root = XMLManager.dict_to_xml(data, root_name)
            tree = ET.ElementTree(root)
            tree.write(filename, encoding='utf-8', xml_declaration=True)
            return f"XML saved to {filename}"
        except Exception as e:
            return f"Error saving XML: {e}"

# Example usage
employee_data = {
    'employee': {
        'personal_info': {
            'name': 'Alice Johnson',
            'age': 30,
            'email': 'alice@company.com'
        },
        'job_info': {
            'position': 'Senior Developer',
            'department': 'Engineering',
            'salary': 95000
        },
        'skills': ['Python', 'JavaScript', 'SQL', 'Docker'],
        'projects': [
            {'name': 'Web API', 'status': 'completed'},
            {'name': 'Mobile App', 'status': 'in_progress'}
        ]
    }
}

xml_manager = XMLManager()
xml_root = xml_manager.dict_to_xml(employee_data, 'company_data')
pretty_xml = xml_manager.prettify_xml(xml_root)

print("XML Output:")
print(pretty_xml)

save_result = xml_manager.save_xml(employee_data, 'employee_data.xml')
print(f"Save result: {save_result}")
```

Part 5: Real-World Applications

1. Web API Data Exchange

python

```python
import json
import requests
from datetime import datetime

class APIClient:
    """Demonstrate serialization in web API context"""

    def __init__(self, base_url):
        self.base_url = base_url
        self.session = requests.Session()

    def create_user(self, user_data):
        """Send user data to API (serialization)"""
        # Prepare data for transmission
        serialized_data = {
            'username': user_data['username'],
            'email': user_data['email'],
            'profile': {
                'first_name': user_data.get('first_name', ''),
                'last_name': user_data.get('last_name', ''),
                'bio': user_data.get('bio', '')
            },
            'preferences': user_data.get('preferences', {}),
            'created_at': datetime.now().isoformat()
        }

        # Serialize to JSON for transmission
        json_payload = json.dumps(serialized_data)

        print("Sending serialized data:")
        print(json.dumps(serialized_data, indent=2))

        # In real scenario, you'd send this to actual API
        # response = self.session.post(f"{self.base_url}/users",
        #                              data=json_payload,
        #                              headers={'Content-Type': 'application/json'})

        # Simulate API response
        simulated_response = {
            'status': 'success',
            'user_id': 12345,
            'message': 'User created successfully',
            'user': serialized_data
```

```
        }

        return simulated_response

    def get_user(self, user_id):
        """Retrieve user data from API (deserialization)"""
        # Simulate API response with JSON data
        json_response = '''
        {
            "user_id": 12345,
            "username": "alice_codes",
            "email": "alice@example.com",
            "profile": {
                "first_name": "Alice",
                "last_name": "Johnson",
                "bio": "Python developer passionate about clean code"
            },
            "preferences": {
                "theme": "dark",
                "notifications": true,
                "language": "en"
            },
            "created_at": "2024-08-16T10:30:00",
            "last_login": "2024-08-16T15:45:00"
        }
        '''

        # Deserialize JSON response to Python object
        user_data = json.loads(json_response)

        print("Received and deserialized data:")
        print(f"User: {user_data['username']}")
        print(f"Email: {user_data['email']}")
        print(f"Full name: {user_data['profile']['first_name']} {user_data['profile']['last_name']}")

        return user_data

# Usage example
api_client = APIClient("https://api.example.com")

# Create user (serialization)
new_user = {
    'username': 'bob_dev',
    'email': 'bob@example.com',
```

```
    'first_name': 'Bob',
    'last_name': 'Smith',
    'bio': 'Full-stack developer',
    'preferences': {
        'theme': 'light',
        'notifications': False
    }
}

create_response = api_client.create_user(new_user)
print("API Response:", create_response['message'])

print("\n" + "="*50 + "\n")

# Get user (deserialization)
user_data = api_client.get_user(12345)
```

2. Configuration Management System

python

```python
import json
import os
from pathlib import Path
from typing import Dict, Any, Optional

class ConfigManager:
    """Advanced configuration management with serialization"""

    def __init__(self, config_dir: str = "config"):
        self.config_dir = Path(config_dir)
        self.config_dir.mkdir(exist_ok=True)
        self.configs: Dict[str, Dict] = {}

    def create_config(self, name: str, config_data: Dict[str, Any]) -> str:
        """Create a new configuration"""
        # Add metadata
        config_with_meta = {
            'metadata': {
                'name': name,
                'version': '1.0',
                'created_at': datetime.now().isoformat(),
                'last_modified': datetime.now().isoformat()
            },
            'config': config_data
        }

        # Save to memory and file
        self.configs[name] = config_with_meta
        return self._save_to_file(name, config_with_meta)

    def update_config(self, name: str, updates: Dict[str, Any]) -> str:
        """Update existing configuration"""
        if name not in self.configs:
            # Try loading from file
            self.load_config(name)

        if name in self.configs:
            # Update the configuration
            self.configs[name]['config'].update(updates)
            self.configs[name]['metadata']['last_modified'] = datetime.now().isoformat()

            # Increment version
            current_version = float(self.configs[name]['metadata']['version'])
```

```
            self.configs[name]['metadata']['version'] = str(current_version + 0.1)

            return self._save_to_file(name, self.configs[name])
        else:
            return f"Configuration '{name}' not found"

    def load_config(self, name: str) -> Optional[Dict]:
        """Load configuration from file"""
        config_file = self.config_dir / f"{name}.json"

        try:
            with open(config_file, 'r') as file:
                config_data = json.load(file)
                self.configs[name] = config_data
                return config_data['config']

        except FileNotFoundError:
            return None
        except json.JSONDecodeError as e:
            print(f"Error loading config '{name}': {e}")
            return None

    def get_config_value(self, name: str, key_path: str, default=None):
        """Get specific config value using dot notation (e.g., 'database.host')"""
        if name not in self.configs:
            self.load_config(name)

        if name not in self.configs:
            return default

        # Navigate through nested keys
        value = self.configs[name]['config']
        for key in key_path.split('.'):
            if isinstance(value, dict) and key in value:
                value = value[key]
            else:
                return default

        return value

    def _save_to_file(self, name: str, config_data: Dict) -> str:
        """Save configuration to JSON file"""
        config_file = self.config_dir / f"{name}.json"
```

```
        try:
            with open(config_file, 'w') as file:
                json.dump(config_data, file, indent=2)
            return f"Configuration '{name}' saved successfully"

        except Exception as e:
            return f"Error saving configuration '{name}': {e}"

    def list_configs(self) -> Dict[str, Dict]:
        """List all available configurations with metadata"""
        configs_info = {}

        # Check files in directory
        for config_file in self.config_dir.glob("*.json"):
            config_name = config_file.stem

            try:
                with open(config_file, 'r') as file:
                    config_data = json.load(file)
                    configs_info[config_name] = config_data.get('metadata', {})
            except:
                configs_info[config_name] = {'status': 'corrupted'}

        return configs_info

    def export_config(self, name: str, export_path: str) -> str:
        """Export configuration to external file"""
        if name not in self.configs:
            self.load_config(name)

        if name in self.configs:
            try:
                with open(export_path, 'w') as file:
                    json.dump(self.configs[name], file, indent=2)
                return f"Configuration exported to {export_path}"
            except Exception as e:
                return f"Export failed: {e}"
        else:
            return f"Configuration '{name}' not found"

# Usage example
config_manager = ConfigManager()

# Create application configuration
```

```
app_config = {
    'database': {
        'host': 'localhost',
        'port': 5432,
        'name': 'myapp',
        'pool_size': 20,
        'timeout': 30
    },
    'cache': {
        'type': 'redis',
        'host': 'localhost',
        'port': 6379,
        'ttl': 3600
    },
    'api': {
        'host': '0.0.0.0',
        'port': 8000,
        'workers': 4,
        'rate_limit': {
            'requests_per_minute': 100,
            'burst_size': 10
        }
    },
    'logging': {
        'level': 'INFO',
        'format': '%(asctime)s - %(name)s - %(levelname)s - %(message)s',
        'file': 'app.log'
    }
}

# Create and save configuration
result = config_manager.create_config('production', app_config)
print(result)

# Update configuration
updates = {
    'database': {
        'host': 'prod-db.example.com',
        'pool_size': 50
    },
    'api': {
        'workers': 8
    }
}
```

```python
update_result = config_manager.update_config('production', updates)
print(update_result)

# Get specific configuration values
db_host = config_manager.get_config_value('production', 'database.host')
api_workers = config_manager.get_config_value('production', 'api.workers')

print(f"Database host: {db_host}")
print(f"API workers: {api_workers}")

# List all configurations
print("\nAvailable configurations:")
for name, metadata in config_manager.list_configs().items():
    print(f"  {name}: version {metadata.get('version', 'unknown')}")
```

Part 6: Advanced Serialization Techniques

Custom Object Serialization

python

```
import json
import pickle
from datetime import datetime, date
from typing import Any, Dict
from dataclasses import dataclass, asdict
from enum import Enum

class Priority(Enum):
    LOW = 1
    MEDIUM = 2
    HIGH = 3
    CRITICAL = 4

@dataclass
class Task:
    """Task class with complex data types"""
    id: int
    title: str
    description: str
    priority: Priority
    created_at: datetime
    due_date: date
    completed: bool = False
    tags: list = None
    metadata: dict = None

    def __post_init__(self):
        if self.tags is None:
            self.tags = []
        if self.metadata is None:
            self.metadata = {}

class TaskSerializer:
    """Custom serialization for Task objects"""

    @staticmethod
    def to_dict(task: Task) -> Dict[str, Any]:
        """Convert Task to dictionary"""
        return {
            'id': task.id,
            'title': task.title,
            'description': task.description,
            'priority': task.priority.value,  # Convert enum to value
```

```
            'created_at': task.created_at.isoformat(),
            'due_date': task.due_date.isoformat(),
            'completed': task.completed,
            'tags': task.tags,
            'metadata': task.metadata,
            '__type__': 'Task'  # Type identifier
        }

    @staticmethod
    def from_dict(data: Dict[str, Any]) -> Task:
        """Convert dictionary to Task"""
        return Task(
            id=data['id'],
            title=data['title'],
            description=data['description'],
            priority=Priority(data['priority']),  # Convert back to enum
            created_at=datetime.fromisoformat(data['created_at']),
            due_date=date.fromisoformat(data['due_date']),
            completed=data['completed'],
            tags=data['tags'],
            metadata=data['metadata']
        )

    @staticmethod
    def to_json(task: Task) -> str:
        """Serialize Task to JSON string"""
        return json.dumps(TaskSerializer.to_dict(task), indent=2)

    @staticmethod
    def from_json(json_str: str) -> Task:
        """Deserialize Task from JSON string"""
        data = json.loads(json_str)
        return TaskSerializer.from_dict(data)

class TaskManager:
    """Task management with serialization"""

    def __init__(self, storage_file: str = "tasks.json"):
        self.storage_file = storage_file
        self.tasks: Dict[int, Task] = {}
        self.load_tasks()

    def add_task(self, title: str, description: str, priority: Priority, due_date: date) -> int:
        """Add a new task"""
```

```
    task_id = len(self.tasks) + 1
    task = Task(
        id=task_id,
        title=title,
        description=description,
        priority=priority,
        created_at=datetime.now(),
        due_date=due_date
    )

    self.tasks[task_id] = task
    self.save_tasks()
    return task_id

def complete_task(self, task_id: int) -> bool:
    """Mark task as completed"""
    if task_id in self.tasks:
        self.tasks[task_id].completed = True
        self.tasks[task_id].metadata['completed_at'] = datetime.now().isoformat()
        self.save_tasks()
        return True
    return False

def add_tag(self, task_id: int, tag: str) -> bool:
    """Add tag to task"""
    if task_id in self.tasks:
        if tag not in self.tasks[task_id].tags:
            self.tasks[task_id].tags.append(tag)
            self.save_tasks()
        return True
    return False

def save_tasks(self) -> None:
    """Save all tasks to JSON file"""
    tasks_data = {
        'tasks': [TaskSerializer.to_dict(task) for task in self.tasks.values()],
        'metadata': {
            'total_tasks': len(self.tasks),
            'last_saved': datetime.now().isoformat(),
            'version': '1.0'
        }
    }

    try:
```

```
            with open(self.storage_file, 'w') as file:
                json.dump(tasks_data, file, indent=2)
        except Exception as e:
            print(f"Error saving tasks: {e}")

    def load_tasks(self) -> None:
        """Load tasks from JSON file"""
        try:
            with open(self.storage_file, 'r') as file:
                data = json.load(file)

                self.tasks = {}
                for task_data in data.get('tasks', []):
                    task = TaskSerializer.from_dict(task_data)
                    self.tasks[task.id] = task

        except FileNotFoundError:
            print(f"No existing task file found. Starting fresh.")
        except Exception as e:
            print(f"Error loading tasks: {e}")

    def get_tasks_by_priority(self, priority: Priority) -> list:
        """Get tasks filtered by priority"""
        return [task for task in self.tasks.values() if task.priority == priority]

    def get_overdue_tasks(self) -> list:
        """Get overdue tasks"""
        today = date.today()
        return [task for task in self.tasks.values()
                if task.due_date < today and not task.completed]

    def export_summary(self, filename: str) -> str:
        """Export task summary to file"""
        summary = {
            'total_tasks': len(self.tasks),
            'completed_tasks': sum(1 for task in self.tasks.values() if task.completed),
            'overdue_tasks': len(self.get_overdue_tasks()),
            'priority_breakdown': {
                'low': len(self.get_tasks_by_priority(Priority.LOW)),
                'medium': len(self.get_tasks_by_priority(Priority.MEDIUM)),
                'high': len(self.get_tasks_by_priority(Priority.HIGH)),
                'critical': len(self.get_tasks_by_priority(Priority.CRITICAL))
            },
            'export_timestamp': datetime.now().isoformat()
```

```
        }

        try:
            with open(filename, 'w') as file:
                json.dump(summary, file, indent=2)
            return f"Summary exported to {filename}"
        except Exception as e:
            return f"Export failed: {e}"

# Demonstration
task_manager = TaskManager()

# Add some tasks
task1_id = task_manager.add_task(
    "Implement user authentication",
    "Add login/logout functionality with JWT tokens",
    Priority.HIGH,
    date(2024, 8, 20)
)

task2_id = task_manager.add_task(
    "Write unit tests",
    "Add comprehensive test coverage for API endpoints",
    Priority.MEDIUM,
    date(2024, 8, 25)
)

task3_id = task_manager.add_task(
    "Fix critical bug",
    "Resolve memory leak in data processing module",
    Priority.CRITICAL,
    date(2024, 8, 17)
)

# Add tags and complete tasks
task_manager.add_tag(task1_id, "authentication")
task_manager.add_tag(task1_id, "security")
task_manager.add_tag(task2_id, "testing")
task_manager.complete_task(task1_id)

# Show task information
print("All tasks:")
for task in task_manager.tasks.values():
    status = "✓ Completed" if task.completed else "○ Pending"
```

```python
    print(f"{status} [{task.priority.name}] {task.title}")
    print(f"    Due: {task.due_date}, Tags: {task.tags}")

print(f"\nOverdue tasks: {len(task_manager.get_overdue_tasks())}")
print(f"High priority tasks: {len(task_manager.get_tasks_by_priority(Priority.HIGH))}")

# Export summary
summary_result = task_manager.export_summary("task_summary.json")
print(f"\n{summary_result}")
```

Part 7: Performance and Security Considerations

Performance Comparison

python

```python
import pickle
import json
import time
import sys
from typing import List, Dict, Any

class PerformanceTester:
    """Compare serialization performance"""

    @staticmethod
    def generate_test_data(size: int) -> Dict[str, Any]:
        """Generate test data of specified size"""
        return {
            'users': [
                {
                    'id': i,
                    'username': f'user_{i}',
                    'email': f'user{i}@example.com',
                    'profile': {
                        'first_name': f'User{i}',
                        'last_name': 'Test',
                        'age': 20 + (i % 50),
                        'preferences': {
                            'theme': 'dark' if i % 2 else 'light',
                            'notifications': i % 3 == 0,
                            'language': 'en'
                        }
                    },
                    'scores': [i * j for j in range(10)],
                    'metadata': {
                        'created_at': f'2024-08-{(i % 28) + 1:02d}T10:00:00',
                        'last_login': f'2024-08-{(i % 28) + 1:02d}T15:30:00'
                    }
                }
                for i in range(size)
            ],
            'metadata': {
                'total_users': size,
                'generated_at': '2024-08-16T12:00:00',
                'version': '1.0'
            }
        }
```

```
    @staticmethod
    def test_json_performance(data: Dict, iterations: int = 100):
        """Test JSON serialization/deserialization performance"""
        print("Testing JSON performance...")

        # Serialization
        start_time = time.time()
        for _ in range(iterations):
            json_string = json.dumps(data)
        serialize_time = time.time() - start_time

        # Deserialization
        start_time = time.time()
        for _ in range(iterations):
            restored_data = json.loads(json_string)
        deserialize_time = time.time() - start_time

        # Size calculation
        json_size = len(json_string.encode('utf-8'))

        return {
            'format': 'JSON',
            'serialize_time': serialize_time,
            'deserialize_time': deserialize_time,
            'total_time': serialize_time + deserialize_time,
            'size_bytes': json_size,
            'size_mb': json_size / (1024 * 1024)
        }

    @staticmethod
    def test_pickle_performance(data: Dict, iterations: int = 100):
        """Test Pickle serialization/deserialization performance"""
        print("Testing Pickle performance...")

        # Serialization
        start_time = time.time()
        for _ in range(iterations):
            pickle_bytes = pickle.dumps(data)
        serialize_time = time.time() - start_time

        # Deserialization
        start_time = time.time()
        for _ in range(iterations):
            restored_data = pickle.loads(pickle_bytes)
```

```
        deserialize_time = time.time() - start_time

        # Size calculation
        pickle_size = len(pickle_bytes)

        return {
            'format': 'Pickle',
            'serialize_time': serialize_time,
            'deserialize_time': deserialize_time,
            'total_time': serialize_time + deserialize_time,
            'size_bytes': pickle_size,
            'size_mb': pickle_size / (1024 * 1024)
        }

    @staticmethod
    def run_performance_comparison(data_sizes: List[int], iterations: int = 100):
        """Run comprehensive performance comparison"""
        print("=" * 80)
        print("SERIALIZATION PERFORMANCE COMPARISON")
        print("=" * 80)

        for size in data_sizes:
            print(f"\nTesting with {size} records...")
            test_data = PerformanceTester.generate_test_data(size)

            # Test JSON
            json_results = PerformanceTester.test_json_performance(test_data, iterations)

            # Test Pickle
            pickle_results = PerformanceTester.test_pickle_performance(test_data, iterations)

            # Display results
            print(f"\nResults for {size} records ({iterations} iterations):")
            print("-" * 60)

            formats = [json_results, pickle_results]
            for result in formats:
                print(f"{result['format']:>8s}: "
                      f"Serialize: {result['serialize_time']:.4f}s, "
                      f"Deserialize: {result['deserialize_time']:.4f}s, "
                      f"Total: {result['total_time']:.4f}s, "
                      f"Size: {result['size_mb']:.2f}MB")

            # Calculate ratios
```

```
        if json_results['total_time'] > 0:
            speed_ratio = pickle_results['total_time'] / json_results['total_time']
            size_ratio = pickle_results['size_mb'] / json_results['size_mb']

            print(f"\nPickle vs JSON:")
            print(f"  Speed ratio: {speed_ratio:.2f}x {'(faster)' if speed_ratio < 1 else '(slower)'}")
            print(f"  Size ratio: {size_ratio:.2f}x {'(smaller)' if size_ratio < 1 else '(larger)'}")

# Run performance tests
tester = PerformanceTester()
tester.run_performance_comparison([100, 500, 1000], iterations=50)
```

Security Considerations

python

```
import pickle
import json
import hashlib
import hmac
from typing import Any, Dict, Optional

class SecureSerializer:
    """Secure serialization with integrity checking"""

    def __init__(self, secret_key: str):
        self.secret_key = secret_key.encode('utf-8')

    def _generate_signature(self, data: bytes) -> str:
        """Generate HMAC signature for data integrity"""
        return hmac.new(self.secret_key, data, hashlib.sha256).hexdigest()

    def _verify_signature(self, data: bytes, signature: str) -> bool:
        """Verify HMAC signature"""
        expected_signature = self._generate_signature(data)
        return hmac.compare_digest(expected_signature, signature)

    def secure_json_serialize(self, data: Dict[str, Any]) -> str:
        """Serialize data with integrity protection"""
        # Serialize the data
        json_data = json.dumps(data, sort_keys=True)  # sort_keys for consistent serialization
        json_bytes = json_data.encode('utf-8')

        # Generate signature
        signature = self._generate_signature(json_bytes)

        # Create secure package
        secure_package = {
            'data': json_data,
            'signature': signature,
            'version': '1.0'
        }

        return json.dumps(secure_package)

    def secure_json_deserialize(self, secure_json: str) -> Optional[Dict[str, Any]]:
        """Deserialize data with integrity verification"""
        try:
            # Parse the secure package
```

```
            secure_package = json.loads(secure_json)

            # Extract components
            data_json = secure_package['data']
            signature = secure_package['signature']

            # Verify signature
            data_bytes = data_json.encode('utf-8')
            if not self._verify_signature(data_bytes, signature):
                raise ValueError("Data integrity check failed - signature mismatch")

            # Deserialize the actual data
            return json.loads(data_json)

        except (json.JSONDecodeError, KeyError, ValueError) as e:
            print(f"Secure deserialization failed: {e}")
            return None

class PickleSafetyDemo:
    """Demonstrate pickle security risks"""

    @staticmethod
    def create_malicious_pickle():
        """Create a pickle that executes code when loaded (DEMO ONLY)"""

        class MaliciousClass:
            def __reduce__(self):
                # This would execute when unpickled (DANGEROUS!)
                return (print, ("⚠️ SECURITY WARNING: This pickle executed code!",))

        return pickle.dumps(MaliciousClass())

    @staticmethod
    def safe_pickle_alternatives():
        """Show safer alternatives to pickle"""
        print("PICKLE SECURITY ALTERNATIVES:")
        print("1. Use JSON when possible (limited data types but safe)")
        print("2. Use protocol buffers for structured data")
        print("3. Use msgpack for binary serialization")
        print("4. Implement custom serialization with validation")
        print("5. Use cryptographic signatures for integrity")

# Security demonstration
print("=" * 60)
```

```
print("SERIALIZATION SECURITY DEMONSTRATION")
print("=" * 60)

# Secure JSON serialization
secure_serializer = SecureSerializer("my_secret_key_12345")

# Test data
sensitive_data = {
    'user_id': 12345,
    'role': 'admin',
    'permissions': ['read', 'write', 'delete'],
    'session_token': 'abc123xyz789',
    'account_balance': 10000.50
}

print("\n1. SECURE JSON SERIALIZATION:")
secure_json = secure_serializer.secure_json_serialize(sensitive_data)
print("✓ Data serialized with integrity protection")

# Verify legitimate deserialization
restored_data = secure_serializer.secure_json_deserialize(secure_json)
if restored_data:
    print("✓ Data verified and restored successfully")
    print(f"  User role: {restored_data['role']}")
    print(f"  Balance: ${restored_data['account_balance']}")
else:
    print("✗ Data verification failed")

# Test tampering detection
print("\n2. TAMPERING DETECTION:")
tampered_json = secure_json.replace('"admin"', '"user"')  # Simulate tampering
tampered_data = secure_serializer.secure_json_deserialize(tampered_json)
if tampered_data is None:
    print("✓ Tampering detected and blocked")
else:
    print("✗ Tampering not detected (security issue)")

# Pickle security warning
print("\n3. PICKLE SECURITY RISKS:")
safety_demo = PickleSafetyDemo()

print("⚠️  WARNING: The following demonstrates why pickle is dangerous:")
malicious_pickle = safety_demo.create_malicious_pickle()
```

```
print("If you uncomment the next line, it will execute malicious code:")
print("# pickle.loads(malicious_pickle)  # DON'T RUN THIS!")

print("\nSafer alternatives:")
safety_demo.safe_pickle_alternatives()
```

Part 8: Practical Implementation - Complete Solution

Now let's implement the requested function and create a comprehensive serialization utility:

python

```python
import json
import os
from pathlib import Path
from typing import Dict, Any, Optional, List, Union
from datetime import datetime

def process_json(data: dict, filename: str) -> dict:
    """
    Process JSON data by serializing to file and deserializing back.

    Args:
        data (dict): Dictionary to serialize
        filename (str): Name of the file to save/load from

    Returns:
        dict: Deserialized dictionary from the file
    """
    try:
        # Ensure filename has .json extension
        if not filename.endswith('.json'):
            filename += '.json'

        # Serialize the dictionary to JSON file
        with open(filename, 'w', encoding='utf-8') as file:
            json.dump(data, file, indent=2, ensure_ascii=False)

        # Deserialize the JSON file back to dictionary
        with open(filename, 'r', encoding='utf-8') as file:
            deserialized_data = json.load(file)

        return deserialized_data

    except Exception as e:
        print(f"Error processing JSON: {e}")
        return {}

class AdvancedSerializationSuite:
    """Complete serialization utility suite"""

    def __init__(self, base_directory: str = "data"):
        self.base_dir = Path(base_directory)
        self.base_dir.mkdir(exist_ok=True)
```

```
    # Create subdirectories for different formats
    (self.base_dir / "json").mkdir(exist_ok=True)
    (self.base_dir / "pickle").mkdir(exist_ok=True)
    (self.base_dir / "backups").mkdir(exist_ok=True)

def save_json(self, data: Dict[str, Any], filename: str,
              create_backup: bool = True) -> str:
    """Enhanced JSON saving with backup and metadata"""
    filepath = self.base_dir / "json" / f"{filename}.json"

    # Create backup if file exists and backup is requested
    if create_backup and filepath.exists():
        backup_name = f"{filename}_backup_{datetime.now().strftime('%Y%m%d_%H%M%S')}.json"
        backup_path = self.base_dir / "backups" / backup_name

        try:
            backup_path.write_text(filepath.read_text())
            backup_msg = f"Backup created: {backup_name}"
        except Exception as e:
            backup_msg = f"Backup failed: {e}"
    else:
        backup_msg = "No backup created"

    # Add metadata to the data
    enhanced_data = {
        'metadata': {
            'filename': filename,
            'created_at': datetime.now().isoformat(),
            'data_type': 'enhanced_json',
            'version': '1.0'
        },
        'payload': data
    }

    try:
        with open(filepath, 'w', encoding='utf-8') as file:
            json.dump(enhanced_data, file, indent=2, ensure_ascii=False)

        return f"✓ JSON saved successfully to {filepath}. {backup_msg}"

    except Exception as e:
        return f"✗ Error saving JSON: {e}"

def load_json(self, filename: str, return_metadata: bool = False) -> Union[Dict, tuple]:
```

```
    """Enhanced JSON loading with metadata handling"""
    filepath = self.base_dir / "json" / f"{filename}.json"

    try:
        with open(filepath, 'r', encoding='utf-8') as file:
            loaded_data = json.load(file)

        # Check if it's enhanced format with metadata
        if 'metadata' in loaded_data and 'payload' in loaded_data:
            if return_metadata:
                return loaded_data['payload'], loaded_data['metadata']
            else:
                return loaded_data['payload']
        else:
            # Legacy format - return as-is
            if return_metadata:
                return loaded_data, {'format': 'legacy'}
            else:
                return loaded_data

    except FileNotFoundError:
        error_msg = f"File {filename}.json not found"
        if return_metadata:
            return {}, {'error': error_msg}
        else:
            print(error_msg)
            return {}
    except Exception as e:
        error_msg = f"Error loading JSON: {e}"
        if return_metadata:
            return {}, {'error': error_msg}
        else:
            print(error_msg)
            return {}

def save_pickle(self, data: Any, filename: str) -> str:
    """Save data using pickle with safety warnings"""
    filepath = self.base_dir / "pickle" / f"{filename}.pkl"

    try:
        with open(filepath, 'wb') as file:
            pickle.dump(data, file)

        return f"✓ Pickle saved to {filepath} (WARNING: Only load from trusted sources!)"
```

```
        except Exception as e:
            return f"✗ Error saving pickle: {e}"

    def load_pickle(self, filename: str, trusted: bool = False) -> Any:
        """Load pickle data with safety check"""
        if not trusted:
            response = input(f"⚠️  Loading pickle file '{filename}.pkl'. "
                             "Only proceed if you trust the source. Continue? (yes/no): ")
            if response.lower() not in ['yes', 'y']:
                print("Pickle loading cancelled for safety.")
                return None

        filepath = self.base_dir / "pickle" / f"{filename}.pkl"

        try:
            with open(filepath, 'rb') as file:
                return pickle.load(file)

        except FileNotFoundError:
            print(f"Pickle file {filename}.pkl not found")
            return None
        except Exception as e:
            print(f"Error loading pickle: {e}")
            return None

    def list_files(self, format_type: str = "all") -> Dict[str, List[str]]:
        """List all saved files by format"""
        files = {'json': [], 'pickle': [], 'backups': []}

        if format_type in ["all", "json"]:
            json_dir = self.base_dir / "json"
            if json_dir.exists():
                files['json'] = [f.stem for f in json_dir.glob("*.json")]

        if format_type in ["all", "pickle"]:
            pickle_dir = self.base_dir / "pickle"
            if pickle_dir.exists():
                files['pickle'] = [f.stem for f in pickle_dir.glob("*.pkl")]

        if format_type in ["all", "backups"]:
            backup_dir = self.base_dir / "backups"
            if backup_dir.exists():
                files['backups'] = [f.stem for f in backup_dir.glob("*.json")]
```

```
        return files

    def get_file_info(self, filename: str, format_type: str) -> Dict[str, Any]:
        """Get detailed information about a saved file"""
        if format_type == "json":
            filepath = self.base_dir / "json" / f"{filename}.json"
        elif format_type == "pickle":
            filepath = self.base_dir / "pickle" / f"{filename}.pkl"
        else:
            return {"error": "Invalid format type"}

        if not filepath.exists():
            return {"error": "File not found"}

        stat = filepath.stat()
        info = {
            'filename': filename,
            'format': format_type,
            'size_bytes': stat.st_size,
            'size_human': f"{stat.st_size / 1024:.2f} KB" if stat.st_size > 1024 else f"{stat.st_size} bytes",
            'created': datetime.fromtimestamp(stat.st_ctime).isoformat(),
            'modified': datetime.fromtimestamp(stat.st_mtime).isoformat()
        }

        # For JSON files, try to get metadata
        if format_type == "json":
            try:
                data, metadata = self.load_json(filename, return_metadata=True)
                info['metadata'] = metadata
            except:
                info['metadata'] = {"error": "Could not read metadata"}

        return info

# Comprehensive demonstration
def main_demonstration():
    """Complete demonstration of serialization concepts"""
    print("=" * 80)
    print("COMPREHENSIVE SERIALIZATION DEMONSTRATION")
    print("=" * 80)

    # Test the required function
    print("\n1. TESTING REQUIRED FUNCTION:")
```

```
sample_data = {
    'name': 'Alice Johnson',
    'age': 30,
    'city': 'Kampala',
    'skills': ['Python', 'JavaScript', 'SQL'],
    'is_employed': True,
    'projects': [
        {'name': 'Web API', 'status': 'completed'},
        {'name': 'Mobile App', 'status': 'in_progress'}
    ]
}

print("Original data:")
print(json.dumps(sample_data, indent=2))

# Use the required function
result = process_json(sample_data, "test_data")

print("\nDeserialized data:")
print(json.dumps(result, indent=2))

print(f"Data preserved correctly: {sample_data == result}")

# Advanced serialization suite demonstration
print("\n2. ADVANCED SERIALIZATION SUITE:")
suite = AdvancedSerializationSuite()

# Test enhanced JSON operations
print("\n2a. Enhanced JSON Operations:")

# Create complex test data
complex_data = {
    'user_profile': {
        'personal_info': {
            'full_name': 'Dr. Sarah Chen',
            'email': 'sarah.chen@university.edu',
            'phone': '+1-555-0123',
            'address': {
                'street': '123 Academic Lane',
                'city': 'Research City',
                'country': 'USA',
                'postal_code': '12345'
            }
        },
```

```
        'professional_info': {
            'position': 'Senior Data Scientist',
            'department': 'AI Research Lab',
            'salary': 125000,
            'start_date': '2020-03-15',
            'skills': ['Machine Learning', 'Python', 'R', 'Statistics'],
            'certifications': [
                'AWS Certified Machine Learning',
                'Google Cloud Professional Data Engineer',
                'Certified Analytics Professional'
            ]
        },
        'project_history': [
            {
                'project_id': 'ML-2024-001',
                'title': 'Predictive Analytics for Healthcare',
                'status': 'completed',
                'budget': 50000,
                'team_size': 5,
                'technologies': ['TensorFlow', 'Python', 'PostgreSQL']
            },
            {
                'project_id': 'AI-2024-002',
                'title': 'Natural Language Processing Pipeline',
                'status': 'in_progress',
                'budget': 75000,
                'team_size': 8,
                'technologies': ['PyTorch', 'BERT', 'Docker', 'Kubernetes']
            }
        ]
    },
    'system_metrics': {
        'performance_data': {
            'cpu_usage': [45.2, 67.8, 34.1, 89.3, 23.7],
            'memory_usage': [78.5, 82.1, 76.9, 91.2, 68.4],
            'disk_io': [156.7, 203.4, 145.8, 298.1, 187.9]
        },
        'error_logs': [
            {'timestamp': '2024-08-16T09:15:32', 'level': 'ERROR', 'message': 'Database connection timeout'},
            {'timestamp': '2024-08-16T10:22:18', 'level': 'WARNING', 'message': 'High memory usage detected'},
            {'timestamp': '2024-08-16T11:45:07', 'level': 'INFO', 'message': 'Backup completed successfully'}
        ]
    }
}
```

```python
# Save with enhanced features
save_result = suite.save_json(complex_data, "user_profile_complex")
print(save_result)

# Load with metadata
loaded_data, metadata = suite.load_json("user_profile_complex", return_metadata=True)
print(f"✓ Data loaded successfully. Metadata: {metadata}")

# Test file information
file_info = suite.get_file_info("user_profile_complex", "json")
print(f"\nFile Information:")
print(f"  Size: {file_info['size_human']}")
print(f"  Created: {file_info['created']}")
print(f"  Format: {file_info['format']}")

print("\n2b. Pickle Operations (Demonstration):")

# Create object that JSON can't handle
class CustomDataProcessor:
    def __init__(self, name, algorithm):
        self.name = name
        self.algorithm = algorithm
        self.data = {'processed_items': 0, 'cache': {}}

    def process_item(self, item):
        self.data['processed_items'] += 1
        self.data['cache'][f'item_{len(self.data["cache"])}'] = item
        return f"Processed {item} using {self.algorithm}"

    def get_stats(self):
        return f"{self.name}: {self.data['processed_items']} items processed"

processor = CustomDataProcessor("DataMiner3000", "Advanced ML Pipeline")
processor.process_item("dataset_1.csv")
processor.process_item("dataset_2.json")
processor.process_item("dataset_3.xml")

print(f"Original processor: {processor.get_stats()}")

# Save to pickle (would require user confirmation in real usage)
pickle_result = suite.save_pickle(processor, "data_processor")
print(pickle_result)
```

```
    # Note: Loading would require user confirmation for safety
    print("Note: Pickle loading requires safety confirmation in interactive mode")

    print("\n2c. File Management:")
    all_files = suite.list_files()
    print("Saved files:")
    for format_type, files in all_files.items():
        if files:
            print(f"  {format_type.upper()}: {', '.join(files)}")

# Real-world use cases
def demonstrate_use_cases():
    """Show practical applications of serialization"""
    print("\n" + "=" * 80)
    print("REAL-WORLD USE CASES")
    print("=" * 80)

    print("\n3. APPLICATION CONFIGURATION MANAGEMENT:")

    # Configuration management example
    app_configs = {
        'development': {
            'database': {
                'host': 'localhost',
                'port': 5432,
                'name': 'myapp_dev',
                'debug': True,
                'pool_size': 5
            },
            'cache': {
                'type': 'memory',
                'ttl': 300
            },
            'logging': {
                'level': 'DEBUG',
                'console': True,
                'file': 'dev.log'
            }
        },
        'production': {
            'database': {
                'host': 'prod-db.company.com',
                'port': 5432,
                'name': 'myapp_prod',
```

```
            'debug': False,
            'pool_size': 50
        },
        'cache': {
            'type': 'redis',
            'host': 'cache.company.com',
            'ttl': 3600
        },
        'logging': {
            'level': 'INFO',
            'console': False,
            'file': '/var/log/myapp.log'
        }
    }
}

suite = AdvancedSerializationSuite("configs")

for env, config in app_configs.items():
    result = suite.save_json(config, f"app_config_{env}")
    print(f"✓ {env.capitalize()} config: {result}")

# Load production config
prod_config = suite.load_json("app_config_production")
print(f"\nProduction database host: {prod_config['database']['host']}")
print(f"Production logging level: {prod_config['logging']['level']}")

print("\n4. USER SESSION MANAGEMENT:")

# Session data example
user_sessions = {
    'session_12345': {
        'user_id': 67890,
        'username': 'alice_codes',
        'login_time': '2024-08-16T09:30:00Z',
        'last_activity': '2024-08-16T11:45:30Z',
        'permissions': ['read', 'write', 'delete'],
        'preferences': {
            'theme': 'dark',
            'language': 'en',
            'timezone': 'UTC-05:00'
        },
        'shopping_cart': [
            {'product_id': 'BOOK-001', 'title': 'Python Mastery', 'price': 49.99, 'quantity': 1},
```

```
            {'product_id': 'COURSE-012', 'title': 'Advanced OOP', 'price': 299.99, 'quantity': 1}
        ],
        'viewed_products': ['BOOK-001', 'BOOK-002', 'COURSE-012', 'COURSE-015']
    }
}

sessions_suite = AdvancedSerializationSuite("sessions")

# Save session data
for session_id, session_data in user_sessions.items():
    result = sessions_suite.save_json(session_data, session_id)
    print(f"Session saved: {session_id}")

# Load and display session info
loaded_session = sessions_suite.load_json('session_12345')
print(f"User: {loaded_session['username']}")
print(f"Cart items: {len(loaded_session['shopping_cart'])}")
print(f"Cart total: ${sum(item['price'] * item['quantity'] for item in loaded_session['shopping_cart'])}")

print("\n5. API RESPONSE CACHING:")

# API response cache example
api_responses = {
    'weather_kampala_20240816': {
        'request_url': 'https://api.weather.com/v1/current?location=kampala',
        'timestamp': '2024-08-16T12:00:00Z',
        'ttl': 1800,  # 30 minutes
        'response_data': {
            'location': 'Kampala, Uganda',
            'temperature': 24,
            'humidity': 78,
            'conditions': 'Partly Cloudy',
            'wind_speed': 12,
            'visibility': 10
        },
        'cache_metadata': {
            'hit_count': 15,
            'last_accessed': '2024-08-16T12:25:00Z'
        }
    },
    'stock_prices_20240816': {
        'request_url': 'https://api.stocks.com/v2/prices?symbols=AAPL,GOOGL,MSFT',
        'timestamp': '2024-08-16T12:00:00Z',
        'ttl': 300,  # 5 minutes
```

```
        'response_data': {
            'AAPL': {'price': 175.32, 'change': +2.15, 'volume': 45678900},
            'GOOGL': {'price': 2847.91, 'change': -12.34, 'volume': 1234567},
            'MSFT': {'price': 342.18, 'change': +5.67, 'volume': 23456789}
        },
        'cache_metadata': {
            'hit_count': 127,
            'last_accessed': '2024-08-16T12:04:30Z'
        }
    }
}

cache_suite = AdvancedSerializationSuite("cache")

for cache_key, cache_data in api_responses.items():
    result = cache_suite.save_json(cache_data, cache_key)
    print(f"Cached: {cache_key}")

# Simulate cache retrieval
weather_cache = cache_suite.load_json('weather_kampala_20240816')
print(f"\nCached weather for {weather_cache['response_data']['location']}:")
print(f"Temperature: {weather_cache['response_data']['temperature']}°C")
print(f"Conditions: {weather_cache['response_data']['conditions']}")
print(f"Cache hits: {weather_cache['cache_metadata']['hit_count']}")

# Best practices and tips
def show_best_practices():
    """Display serialization best practices"""
    print("\n" + "=" * 80)
    print("SERIALIZATION BEST PRACTICES & TIPS")
    print("=" * 80)

    practices = [
        {
            'category': 'Format Selection',
            'tips': [
                'Use JSON for web APIs and configuration files (human-readable)',
                'Use Pickle only for Python-to-Python communication (feature-rich)',
                'Use MessagePack for high-performance binary serialization',
                'Use Protocol Buffers for cross-language data exchange',
                'Use YAML for configuration files that humans need to edit'
            ]
        },
        {
```

```
        'category': 'Security',
        'tips': [
            'NEVER unpickle data from untrusted sources',
            'Use HMAC signatures for data integrity verification',
            'Validate and sanitize deserialized data',
            'Consider encryption for sensitive data',
            'Implement version checking for data compatibility'
        ]
    },
    {
        'category': 'Performance',
        'tips': [
            'Use compression for large datasets (gzip, lz4)',
            'Cache frequently accessed serialized data',
            'Stream large datasets instead of loading entirely',
            'Use binary formats for performance-critical applications',
            'Profile different formats for your specific use case'
        ]
    },
    {
        'category': 'Data Management',
        'tips': [
            'Include metadata (version, timestamp, schema)',
            'Implement backward compatibility strategies',
            'Use consistent field naming conventions',
            'Handle missing or null values gracefully',
            'Maintain separate development and production data'
        ]
    },
    {
        'category': 'Error Handling',
        'tips': [
            'Always wrap serialization in try-catch blocks',
            'Provide meaningful error messages',
            'Implement fallback strategies for corrupted data',
            'Log serialization/deserialization operations',
            'Test with various data types and edge cases'
        ]
    }
]

for practice in practices:
    print(f"\n{practice['category'].upper()}:")
    for i, tip in enumerate(practice['tips'], 1):
```

```
        print(f"  {i}. {tip}")

# Run all demonstrations
if __name__ == "__main__":
    main_demonstration()
    demonstrate_use_cases()
    show_best_practices()

    print("\n" + "=" * 80)
    print("SERIALIZATION MASTERY COMPLETE! 🎉")
    print("=" * 80)
    print("You now understand:")
    print("✓ JSON serialization for web APIs and configs")
    print("✓ Pickle serialization for Python objects")
    print("✓ Security considerations and best practices")
    print("✓ Performance optimization techniques")
    print("✓ Real-world application patterns")
    print("✓ Error handling and data integrity")
    print("\nNext steps: Practice with your own data and use cases!")

# Example usage of the required function
def test_process_json_function():
    """Test the specific function requested in the task"""
    print("\n" + "="*50)
    print("TESTING PROCESS_JSON FUNCTION")
    print("="*50)

    # Test case 1: Simple data
    simple_data = {
        'name': 'Alice',
        'age': 30,
        'city': 'Kampala'
    }

    result1 = process_json(simple_data, 'test_simple')
    print("Test 1 - Simple data:")
    print(f"Original: {simple_data}")
    print(f"Result: {result1}")
    print(f"Match: {simple_data == result1}")

    # Test case 2: Complex nested data
    complex_data = {
        'user': {
            'personal': {
```

```
            'name': 'Bob Smith',
            'age': 25,
            'contacts': ['email@example.com', '+256-123-456789']
        },
        'professional': {
            'title': 'Software Developer',
            'skills': ['Python', 'JavaScript', 'SQL'],
            'experience': 3
        }
    },
    'metadata': {
        'created': '2024-08-16',
        'version': 1.2,
        'active': True
    }
}

result2 = process_json(complex_data, 'test_complex')
print("\nTest 2 - Complex data:")
print("✓ Data processed successfully")
print(f"Match: {complex_data == result2}")

# Test case 3: Edge cases
edge_case_data = {
    'empty_list': [],
    'empty_dict': {},
    'null_value': None,
    'boolean_true': True,
    'boolean_false': False,
    'zero': 0,
    'negative': -42,
    'float': 3.14159,
    'unicode': 'Hello 世界 🌍'
}

result3 = process_json(edge_case_data, 'test_edge_cases')
print("\nTest 3 - Edge cases:")
print("✓ Edge cases handled successfully")
print(f"Match: {edge_case_data == result3}")

return result1, result2, result3
```

```
# Run the specific test
test_results = test_process_json_function()
```