# Day 1, Session 2: Data Structures and Numerical Computing

## Three-Day Data Analysis with Python Course

## Session Overview

1. **Core Data Structures**
2. **Introduction to NumPy Arrays**
3. **Working with Files**

# Part 1: Core Data Structures

## Core Data Structures

Python provides several built-in data structures for organizing and storing data.

Each has different characteristics and use cases.

# Lists

**Ordered, mutable collections** of items.

**Key characteristics:**

- Ordered (maintains insertion order)
- Mutable (can be changed after creation)
- Allows duplicates
- Indexed by position (0, 1, 2, ...)

**Syntax:**

```
my_list = [item1, item2, item3]
my_list[0]  # Access first item
```

# Lists (Example)

```
# Creating lists
sales = [1200, 1450, 1100, 1800, 2100]
products = ["Laptop", "Mouse", "Keyboard", "Monitor"]
mixed = [100, "Product", True, 3.14]

print(f"Sales: {sales}")
print(f"Products: {products}")
print(f"Mixed types: {mixed}")

# Accessing elements
print(f"\nFirst sale: ${sales[0]}")
print(f"Last product: {products[-1]}")  # Negative indexing
print(f"First three sales: {sales[0:3]}")  # Slicing
```

# List Methods

Common operations for modifying lists:

| Method | Description | Example |
|---|---|---|
| `.append(x)` | Add item to end | `list.append(5)` |
| `.insert(i, x)` | Insert at position | `list.insert(0, 5)` |
| `.remove(x)` | Remove first occurrence | `list.remove(5)` |
| `.pop()` | Remove and return last item | `list.pop()` |
| `.sort()` | Sort in place | `list.sort()` |
| `.reverse()` | Reverse in place | `list.reverse()` |

# List Methods (Example)

```
# Start with a list of daily sales
daily_sales = [1200, 1450, 1100]

# Add new sales
daily_sales.append(1800)
print(f"After append: {daily_sales}")

# Insert at specific position
daily_sales.insert(0, 1000)
print(f"After insert: {daily_sales}")

# Remove a value
daily_sales.remove(1100)
print(f"After remove: {daily_sales}")
```

# List Methods (Continued)

```
# Sort the list
daily_sales.sort()
print(f"After sort: {daily_sales}")

# Get list statistics
print(f"\nTotal sales: ${sum(daily_sales):,}")
print(f"Number of days: {len(daily_sales)}")
print(f"Average: ${sum(daily_sales) / len(daily_sales):.2f}")
```

# Tuples

**Ordered, immutable collections** of items.

**Key characteristics:**

- Ordered (maintains insertion order)
- **Immutable** (cannot be changed after creation)
- Allows duplicates

**Use cases:**

- Fixed collections (coordinates, RGB colors)
- Function return values

# Tuples (Syntax)

## Syntax:

```
my_tuple = (item1, item2, item3)
my_tuple[0]  # Access first item
```

# Dictionaries

**Unordered collections of key-value pairs**.

**Key characteristics:**

- Keys must be unique and immutable (strings, numbers, tuples)
- Values can be any type
- Mutable

**Use cases:**

- Representing structured data (like JSON)
- Mapping relationships

# Dictionaries (Syntax)

## Syntax:

```
my_dict = {"key1": value1, "key2": value2}
my_dict["key1"]  # Access value by key
```

# Dictionaries (Example)

```
# Creating dictionaries - representing a customer
customer = {
    "id": 12345,
    "name": "Alice Johnson",
    "email": "alice@example.com",
    "purchases": 15,
    "total_spent": 2450.00
}

print("Customer information:")
print(f"Name: {customer['name']}")
print(f"Email: {customer['email']}")
print(f"Total spent: ${customer['total_spent']:.2f}")
```

# Dictionaries (Continued)

```
# Adding new key-value pairs
customer["vip_status"] = True
print(f"\nVIP Status: {customer['vip_status']}")

# Updating values
customer["purchases"] += 1
customer["total_spent"] += 150.00
print(f"Updated purchases: {customer['purchases']}")
print(f"Updated total: ${customer['total_spent']:.2f}")
```

# Dictionary Methods

Essential dictionary operations:

| Method | Description | Example |
|---|---|---|
| `.keys()` | Get all keys | `dict.keys()` |
| `.values()` | Get all values | `dict.values()` |
| `.items()` | Get key-value pairs | `dict.items()` |
| `.get(key, default)` | Safe access | `dict.get("age", 0)` |
| `.update(other)` | Merge dictionaries | `dict.update({...})` |

# Dictionary Methods (Example)

```
# Product inventory
inventory = {
    "laptop": 45,
    "mouse": 120,
    "keyboard": 78,
    "monitor": 32
}

# Iterating over keys
print("Products in stock:")
for product in inventory.keys():
    print(f"  - {product}")

# Iterating over values
print(f"\nTotal items: {sum(inventory.values())}")
```

# Dictionary Methods (Continued)

```
# Iterating over key-value pairs
print("\nFull inventory:")
for product, quantity in inventory.items():
    print(f"  {product.capitalize()}: {quantity} units")

# Safe access with .get()
tablets = inventory.get("tablet", 0)
print(f"\nTablets in stock: {tablets}")

# Practical: find low stock items
print("\nLow stock alerts (< 50 units):")
for product, quantity in inventory.items():
    if quantity < 50:
        print(f"  ⚠️  {product}: only {quantity} left!")
```

# Sets

**Unordered collections of unique items**.

**Key characteristics:**

- No duplicates (automatically removed)
- Unordered (no indexing)
- Mutable

**Use cases:**

- Removing duplicates
- Set operations (union, intersection, difference)
- Membership testing

# Sets (Syntax)

## Syntax:

```
my_set = {item1, item2, item3}
item in my_set  # Check membership
```

# When to Use Each Structure?

| Structure | Use When... |
|---|---|
| **List** | Ordered collection, need indexing, duplicates OK |
| **Dictionary** | Key-value mapping, fast lookup by key |
| Tuple | Immutable collection, fixed data |
| Set | Unique items, set operations, membership testing |

# Part 2: Introduction to NumPy Arrays

# Why NumPy?

**NumPy** is the go-to library for numerical computing in Python.

- Much faster than Python lists for numerical operations
- Memory efficient
- Supports vectorized operations
- Foundation for pandas, scipy, and many other libraries

# Importing NumPy

```
# Import NumPy (standard alias is 'np')
import numpy as np

print(f"NumPy version: {np.__version__}")
```

## NumPy Arrays

Arrays are the core data structure in NumPy.

- An array is a list where every item has the same numerical type
- The length of the array is also fixed
- Arrays can be multidimensional (matrices, tensors)

# Creating Arrays (Example)

```
# Create array from a list
sales_list = [1200, 1450, 1100, 1800, 2100, 950, 1350]
sales_array = np.array(sales_list)
print(f"Sales array: {sales_array}")
print(f"Type: {type(sales_array)}")
print(f"Data type: {sales_array.dtype}")

# Create arrays with NumPy functions
zeros = np.zeros(5)
ones = np.ones(5)
sequence = np.arange(1, 8)  # Like range(), but returns array

print(f"\nZeros: {zeros}")
print(f"Ones: {ones}")
print(f"Sequence: {sequence}")
```

## Array Operations

NumPy arrays support vectorized operations - operations on entire arrays without loops!

**Much faster than Python lists for numerical work.**

# Vectorized Operations (Example)

```
# Arithmetic operations
daily_sales = np.array([1200, 1450, 1100, 1800, 2100])

# Apply 10% discount to all sales (vectorized!)
discounted_sales = daily_sales * 0.9
print(f"Original: {daily_sales}")
print(f"After 10% discount: {discounted_sales}")

# Add a bonus to all sales
with_bonus = daily_sales + 100
print(f"With $100 bonus: {with_bonus}")
```

# Array Statistics

NumPy provides efficient statistical functions:

| Function | Description |
|---|---|
| `np.mean()` | Average |
| `np.median()` | Median |
| `np.std()` | Standard deviation |
| `np.min()` | Minimum |
| `np.max()` | Maximum |
| `np.sum()` | Sum |

# Statistical Analysis (Example)

```
# Sales data for analysis
weekly_sales = np.array([1200, 1450, 1100, 1800, 2100, 950, 1350])

print("Weekly Sales Analysis")
print("=" * 40)
print(f"Total revenue: ${np.sum(weekly_sales):,.2f}")
print(f"Average daily sales: ${np.mean(weekly_sales):,.2f}")
print(f"Median: ${np.median(weekly_sales):,.2f}")
print(f"Std deviation: ${np.std(weekly_sales):,.2f}")
print(f"Minimum: ${np.min(weekly_sales):,.2f}")
print(f"Maximum: ${np.max(weekly_sales):,.2f}")
print(f"Range: ${np.max(weekly_sales) - np.min(weekly_sales):,.2f}")
```

# Finding Indices

```
# Find indices
best_day = np.argmax(weekly_sales)
worst_day = np.argmin(weekly_sales)
print(f"\nBest day: Day {best_day + 1} (${weekly_sales[best_day]})")
print(f"Worst day: Day {worst_day + 1} (${weekly_sales[worst_day]})")
```

# Lists vs Arrays: Quick Comparison

**Use Python lists when:**

- Mixed data types
- Frequent insertions/deletions
- Small datasets

**Use NumPy arrays for:**

- Numerical computations
- Need vectorized operations
- Statistical analysis

**Performance difference can be 10-100x for large datasets!**

# Part 3: Working with Files

## Working with Files

Real-world data comes from files, so we need to process them.

**Common file formats:**

- Text files (.txt)
- CSV files (.csv)
- JSON files (.json)

# Reading and Writing Text Files

**Syntax:**

```
with open("filename.txt", "r") as file:
    content = file.read()
```

**Modes:**

- `"r"` - Read
- `"w"` - Write (overwrites)
- `"a"` - Append

# Text File Operations (Example)

```
# Writing to a file
sales_report = """Daily Sales Report
==================
Day 1: $1,200
Day 2: $1,450
Day 3: $1,100
Day 4: $1,800
Day 5: $2,100
"""

with open("sales_report.txt", "w") as file:
    file.write(sales_report)

print("✓ File written: sales_report.txt")
```

# Reading Text Files

```
# Reading from a file
with open("sales_report.txt", "r") as file:
    content = file.read()

print("\nFile contents:")
print(content)

# Reading line by line
print("Reading line by line:")
with open("sales_report.txt", "r") as file:
    for line_num, line in enumerate(file, 1):
        print(f"  Line {line_num}: {line.strip()}")
```

# Working with CSV Files

Python's `csv` module makes it easy to work with CSV files.

**Structure:**

```
name,age,city
Alice,30,New York
Bob,25,San Francisco
```

# CSV Operations (Example)

```
import csv

# Writing CSV
customers = [
    ["id", "name", "purchases", "total"],
    [101, "Alice Johnson", 15, 2450.00],
    [102, "Bob Smith", 8, 1320.50],
    [103, "Carol White", 22, 3890.00],
    [104, "David Brown", 5, 780.00]
]

with open("customers.csv", "w", newline="") as file:
    writer = csv.writer(file)
    writer.writerows(customers)

print("✓ CSV file written: customers.csv")
```

# Reading CSV Files

```
# Reading CSV
print("\nReading CSV file:")
with open("customers.csv", "r") as file:
    reader = csv.reader(file)
    for row in reader:
        print(f"  {row}")

# Reading CSV as dictionaries (more convenient!)
print("\nReading CSV as dictionaries:")
with open("customers.csv", "r") as file:
    dict_reader = csv.DictReader(file)
    for row in dict_reader:
        print(f"  Customer {row['id']}: {row['name']} - ${row['total']}")
```

## Working with JSON Files

**JSON** is perfect for structured data.

**Python's `json` module** handles serialization and deserialization.

# JSON Operations (Example)

```
import json

# Python dictionary
customer_data = {
    "id": 12345,
    "name": "Alice Johnson",
    "email": "alice@example.com",
    "purchases": [
        {"date": "2024-01-15", "amount": 150.00, "product": "Laptop"},
        {"date": "2024-02-20", "amount": 50.00, "product": "Mouse"},
        {"date": "2024-03-10", "amount": 85.00, "product": "Keyboard"}
    ],
    "vip": True
}
```

# JSON Operations (Continued)

```
# Writing JSON
with open("customer.json", "w") as file:
    json.dump(customer_data, file, indent=2)

print("✓ JSON file written: customer.json")
```

# Reading JSON Files

```
# Reading JSON
with open("customer.json", "r") as file:
    loaded_data = json.load(file)

print("\nLoaded customer data:")
print(f"Name: {loaded_data['name']}")
print(f"Email: {loaded_data['email']}")
print(f"VIP Status: {loaded_data['vip']}")
print(f"\nPurchases:")
for purchase in loaded_data['purchases']:
    print(f"  {purchase['date']}: {purchase['product']} - ${purchase['amount']:.2f}")
```

# Session 2 Summary

## What We Covered

- ✓ **Data Structures**: Lists, tuples, dictionaries, sets
- ✓ **NumPy Arrays**: Vectorized operations and statistics
- ✓ **File Operations**: Reading/writing text, CSV, and JSON

# Questions?

**Lunch break!** 🍕

See you for the practical session!