# Python Data Structures: Complete Guide

## 1. Tuples and Sequences

Tuples are immutable sequences of items enclosed in parentheses.

### When to Use Tuples:

- When you need an **immutable** collection of items
- For **returning multiple values** from a function
- As **dictionary keys** (since they're hashable)
- For **data that shouldn't change** (like coordinates)
- When you want to ensure data **integrity**
- For **heterogeneous data** that forms a logical record

### Example:

```python
def transpose_matrix(matrix):
    """
    Transpose a matrix represented as a tuple of tuples.
    Args:
        matrix: Tuple of tuples (rows of the matrix)
    Returns:
        Transposed matrix as a tuple of tuples
    """
    # Check if matrix is empty
    if not matrix or not matrix[0]:
        return ()
    # Use zip to transpose the matrix
    transposed = tuple(zip(*matrix))
    return transposed

# Test case
original = ((1, 2, 3), (4, 5, 6))
transposed = transpose_matrix(original)
print(f"Original: {original}")      # ((1, 2, 3), (4, 5, 6))
print(f"Transposed: {transposed}")  # ((1, 4), (2, 5), (3, 6))
```

## 2. Sets

Sets are unordered collections of unique elements.

### When to Use Sets:

- When you need to store **unique values** without duplicates
- For **mathematical set operations** (union, intersection, difference)
- When you need **fast membership testing** (O(1) complexity)
- When **order doesn't matter**
- For **removing duplicates** from a sequence
- When you need to determine **commonality** between collections

### Example:

python

```python
def largest_distance(coordinates):
    """
    Find the coordinate with the largest distance from origin (0,0).
    Args:
        coordinates: List of (x, y) tuples
    Returns:
        Tuple with largest distance from origin
    """
    if not coordinates:
        return None
    # Function to calculate Euclidean distance
    def distance(point):
        x, y = point
        return (x**2 + y**2)**0.5
    # Find the point with maximum distance
    return max(coordinates, key=distance)

# Test cases
test_coordinates = [(1, 1), (3, 4), (-2, 2), (0, 5)]
print(largest_distance(test_coordinates))  # Should return (3, 4) with distance 5

def analyze_lists(list1, list2):
    """
    Analyze two lists and return common elements and elements unique to each list.
    Args:
        list1: First list
        list2: Second list
    Returns:
        Tuple containing (common elements, unique to list1, unique to list2)
    """
    set1 = set(list1)
    set2 = set(list2)
    common = set1 & set2
    only_in_list1 = set1 - set2
    only_in_list2 = set2 - set1
    return (common, only_in_list1, only_in_list2)

# Test case
fruits1 = ["apple", "banana", "cherry", "date"]
fruits2 = ["banana", "date", "elderberry", "fig"]
common, only1, only2 = analyze_lists(fruits1, fruits2)
print(f"Common: {common}")          # {'banana', 'date'}
```

```python
print(f"Only in list1: {only1}")     # {'apple', 'cherry'}
print(f"Only in list2: {only2}")     # {'elderberry', 'fig'}
```

## 3. Dictionaries

Dictionaries are mutable mappings that store key-value pairs.

### When to Use Dictionaries:

- When you need **key-value associations**
- For **fast lookups** by key (O(1) complexity)
- When counting **frequencies** or occurrences
- For **grouping/categorizing** data
- When implementing **caches** or memoization
- For **representing objects** with named attributes
- For **sparse data** storage
- When you need to **map one value to another**

### Example:

python

```python
def character_frequency(text):
    """
    Count the frequency of each character in a string.
    Args:
        text: Input string
    Returns:
        Dictionary mapping characters to their frequencies
    """
    frequency = {}
    for char in text:
        if char in frequency:
            frequency[char] += 1
        else:
            frequency[char] = 1
    # Alternative using get method:
    # for char in text:
    #     frequency[char] = frequency.get(char, 0) + 1
    return frequency

# Test case
text = "hello world"
print(character_frequency(text))
# {'h': 1, 'e': 1, 'l': 3, 'o': 2, ' ': 1, 'w': 1, 'r': 1, 'd': 1}

def index_map(sequence):
    """
    Create a mapping from elements to their indices in the sequence.
    Args:
        sequence: Input sequence (list, tuple, or string)
    Returns:
        Dictionary mapping elements to lists of indices
    """
    result = {}
    for i, element in enumerate(sequence):
        if element in result:
            result[element].append(i)
        else:
            result[element] = [i]
    return result

# Test cases
test_list = [1, 2, 3, 2, 1, 4, 5, 2]
print(index_map(test_list))
```

```python
# {1: [0, 4], 2: [1, 3, 7], 3: [2], 4: [5], 5: [6]}
test_string = "hello"
print(index_map(test_string))
# {'h': [0], 'e': [1], 'l': [2, 3], 'o': [4]}
```

## 4. Lists

Lists are mutable sequences that can contain items of different types.

### When to Use Lists:

- When you need a **mutable, ordered** collection
- When you need to **append or remove** elements frequently
- When **indexing** is important
- When you need to **store duplicates**
- For **stack or queue** implementations
- When **order matters** and items need to be **rearranged**
- For **temporary collections** that will be modified

### Example:

python

```python
def longest_common_subsequence(seq1, seq2):
    """
    Find the longest common subsequence between two sequences.
    Args:
        seq1: First sequence (string, list, or tuple)
        seq2: Second sequence (string, list, or tuple)
    Returns:
        Longest common subsequence as the same type as the first input
    """
    # Create a table to store lengths of LCS
    m, n = len(seq1), len(seq2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]
    # Fill the dp table
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if seq1[i-1] == seq2[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])
    # Backtrack to find the LCS
    i, j = m, n
    lcs = []
    while i > 0 and j > 0:
        if seq1[i-1] == seq2[j-1]:
            lcs.append(seq1[i-1])
            i -= 1
            j -= 1
        elif dp[i-1][j] > dp[i][j-1]:
            i -= 1
        else:
            j -= 1
    # Reverse the list to get correct order
    lcs.reverse()
    # Convert result to the same type as the first input
    if isinstance(seq1, str):
        return ''.join(lcs)
    elif isinstance(seq1, tuple):
        return tuple(lcs)
    else:  # List or other sequence
        return lcs

# Test cases
print(longest_common_subsequence("ABCDEF", "ACBCF"))  # "ABCF"
```

```python
print(longest_common_subsequence([1, 2, 3, 4, 5], [1, 3, 5, 7]))  # [1, 3, 5]
print(longest_common_subsequence((1, 2, 3), (2, 3, 4)))  # (2, 3)
```

## 5. Nested Data Structures

Combining data structures allows for complex data representation and manipulation.

### When to Use Nested Data Structures:

- For **hierarchical data** representation
- When dealing with **complex relationships**
- For **multi-dimensional** data
- When implementing **graphs** or **trees**
- For **advanced categorization** and organization

### Example:

python

```python
def organize_students(students):
    """
    Organize student data by grade.
    Args:
        students: List of dictionaries with 'name', 'grade', and subject scores
    Returns:
        Nested dictionary: {grade: {name: average_score}}
    """
    result = {}
    for student in students:
        name = student['name']
        grade = student['grade']
        # Calculate average score (exclude name and grade from calculation)
        scores = [value for key, value in student.items()
                if key not in ('name', 'grade')]
        average = sum(scores) / len(scores) if scores else 0
        # Add to the nested dictionary structure
        if grade not in result:
            result[grade] = {}
        result[grade][name] = average
    return result

# Test case
students_data = [
    {'name': 'Alice', 'grade': 'A', 'math': 95, 'science': 90, 'english': 92},
    {'name': 'Bob', 'grade': 'B', 'math': 80, 'science': 85, 'english': 75},
    {'name': 'Charlie', 'grade': 'A', 'math': 98, 'science': 96, 'english': 94}
]
organized = organize_students(students_data)
print(organized)
# {'A': {'Alice': 92.33333333333333, 'Charlie': 96.0}, 'B': {'Bob': 80.0}}

def analyze_skills(people):
    """
    Analyze skills across people.
    Args:
        people: List of dictionaries, each with 'name' and 'skills' keys
    Returns:
        Tuple containing (set of all skills, dictionary mapping skills to people)
    """
    all_skills = set()
    skill_to_people = {}
    for person in people:
```

```python
        name = person['name']
        skills = person['skills']
        # Add to the set of all skills
        all_skills.update(skills)
        # Add person to each skill's list
        for skill in skills:
            if skill in skill_to_people:
                skill_to_people[skill].append(name)
            else:
                skill_to_people[skill] = [name]
    return (all_skills, skill_to_people)

# Test case
people_data = [
    {'name': 'Alice', 'skills': ['Python', 'SQL', 'JavaScript']},
    {'name': 'Bob', 'skills': ['Java', 'C++', 'Python']},
    {'name': 'Charlie', 'skills': ['JavaScript', 'HTML', 'CSS']}
]
all_skills, skill_map = analyze_skills(people_data)
print(f"All skills: {all_skills}")
print(f"Skill to people mapping: {skill_map}")
```

## 6. Data Structure Selection Guide

### Choose Tuples When:

- You need **immutable data**
- You want to **prevent accidental modification**
- You're using the collection as a **dictionary key** or **set element**
- You have **fixed data** like coordinates or RGB values

### Choose Sets When:

- You need **unique elements**
- You want to **eliminate duplicates** quickly
- You need **set operations** (union, intersection, difference)
- You need **fast membership tests** (is x in collection?)
- **Order is not important**

### Choose Dictionaries When:

- You need **key-value pairs**

- You want **fast lookups by key**
- You're **counting frequencies**
- You're **grouping data** by some attribute
- You're implementing a **cache** or **lookup table**

### Choose Lists When:

- You need an **ordered collection**
- You need a **mutable sequence**
- You want to **frequently modify** the collection
- You need to **preserve duplicates**
- You need to **sort items** or **access by index**

### Performance Considerations:

| Operation | List | Tuple | Dictionary | Set |
|---|---|---|---|---|
| Access by index | O(1) | O(1) | N/A | N/A |
| Insert/Delete | O(n) | N/A | O(1) | O(1) |
| Append | O(1) | N/A | N/A | N/A |
| Membership Test | O(n) | O(n) | O(1) | O(1) |
| Iteration | O(n) | O(n) | O(n) | O(n) |
| Length | O(1) | O(1) | O(1) | O(1) |

This performance guide can help you choose the most efficient data structure for your specific needs.