Department of COMPUTER SCIENCE AND DESIGN Python Module 2 VTU Notes: Lists, Tuples & Dictionaries

Abhishek Patil Department of CSD

Module 2 Overview: Core Python Data Structures

Understand Fundamentals

Master Python's three essential data structures: lists for ordered sequences, tuples for immutable data, and dictionaries for key-value mappings. Each structure serves distinct purposes in programming.

Learn Selection Criteria

Discover when to choose each structure based on mutability needs, performance requirements, and data relationships. Understanding these properties enables optimal design decisions.

Apply to Real Problems

Transform theoretical knowledge into practical solutions by implementing these structures in algorithms, data processing tasks, and application development scenarios.

This module forms the foundation for advanced Python programming, enabling you to build sophisticated applications, process complex datasets, and write efficient, maintainable code. By the end of this module, you'll confidently select and manipulate the appropriate data structure for any programming challenge you encounter.

Lists: Ordered, Mutable Collections

Lists are Python's most versatile data structure, providing ordered collections that can grow, shrink, and change dynamically during program execution. They're created using square brackets and can hold any combination of data types.

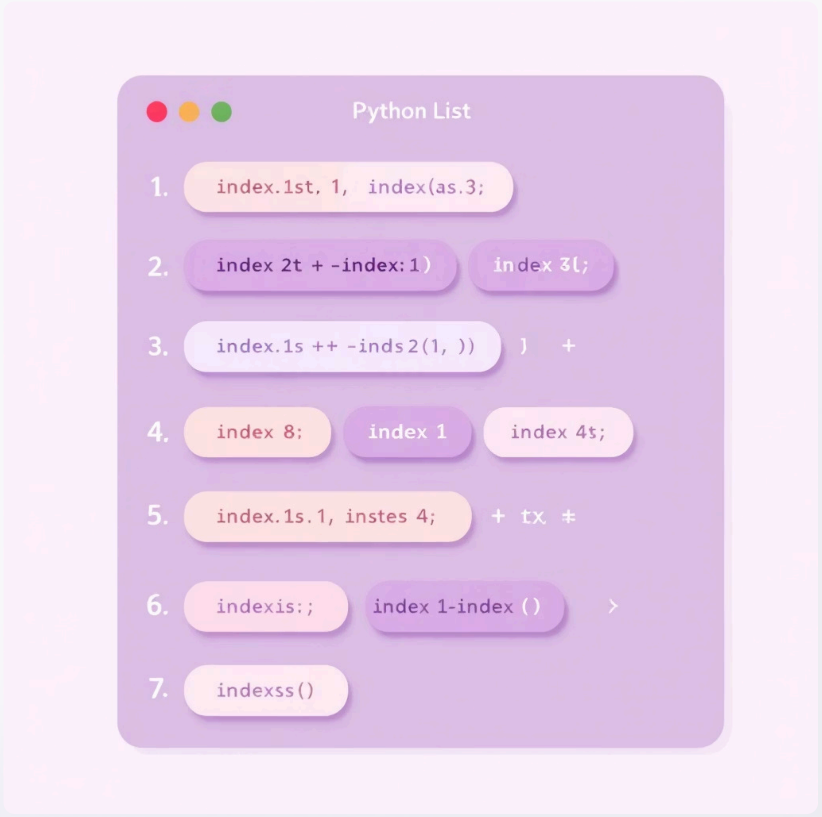
Key Characteristics

- **Ordered sequences**: Items maintain their insertion order and can be accessed by position
- **Heterogeneous elements**: Mix strings, integers, floats, booleans, and even other lists
- **Mutable nature**: Add, remove, or modify elements after creation without creating a new list
- **Dynamic sizing**: Lists automatically adjust their size as elements are added or removed
- **Allow duplicates**: The same value can appear multiple times in different positions

Syntax Examples

```
# Creating lists
fruits = ['apple', 'banana', 'cherry']
mixed = [1, 'hello', 3.14, True]
nested = [[1, 2], [3, 4], [5, 6]]
empty = []
```

Lists are ideal when you need a collection that changes frequently, such as tracking items in a shopping cart, maintaining a to-do list, or storing user inputs during program execution.


Python List
1. index.1st, 1, index(as.3;
2. index 2t + -index: 1) index 3(;
3. index.1s ++ -inds 2(1,))) +
4. index 8: index 1 index 4s;
5. index.1s. 1, instes 4; + tx ≠
6. indexis:; index 1-index () ›
7. indexss()

💡 Memory Tip

Think of lists as **expandable boxes** where you can add, remove, or rearrange items freely. The square brackets [] symbolize a container that holds your data.

Accessing List Elements & Indexing

Python provides flexible indexing mechanisms that allow both forward and backward traversal of lists. Understanding indexing is crucial for manipulating list data effectively.

Zero-Based Positive Indexing

Python lists start counting from 0, not 1. The first element is at index 0, the second at index 1, and so on.

```
spam = ['cat', 'bat', 'rat', 'elephant']
print(spam[0])  # Output: 'cat'
print(spam[2])  # Output: 'rat'
```

Negative Indexing

Access elements from the end using negative indices. `-1` refers to the last item, `-2` to the second-last, and so forth.

```
print(spam[-1])  # Output: 'elephant'
print(spam[-3])  # Output: 'bat'
```

Nested List Access

For lists within lists, use multiple bracket pairs to access inner elements through successive indexing operations.

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(matrix[0][1])  # Output: 2
print(matrix[2][0])  # Output: 7
```

Common Indexing Errors

IndexError: Occurs when accessing an index beyond the list's length. **TypeError**: Happens when using non-integer indices like floats or strings.

```
spam[10] # IndexError: list index out of range
spam['two'] # TypeError: list indices must be integers
```

List Methods & Operations

Essential List Methods

Python provides powerful built-in methods that make list manipulation intuitive and efficient. These methods modify lists in-place or return new values.

.append(item)

Adds a single element to the end of the list

```
fruits.append('mango')
```

.remove(item)

Removes the first occurrence of the specified value

```
fruits.remove('apple')
```

.pop(index)

Removes and returns item at index (default: last item)

```
last = fruits.pop()
```

.sort()

Sorts the list in ascending order in-place

```
numbers.sort()
```

.reverse()

Reverses the order of elements in-place

```
fruits.reverse()
```

Advanced Operations

List Slicing

Extract sublists using the [start:stop:step] syntax. Slicing creates new lists without modifying the original.

```
spam = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
spam[1:4]   # [1, 2, 3]
spam[:5]    # [0, 1, 2, 3, 4]
spam[5:]    # [5, 6, 7, 8, 9]
spam[::2]   # [0, 2, 4, 6, 8]
spam[::-1]  # [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

List Comprehensions

Create new lists elegantly using concise, readable syntax that combines loops and conditions.

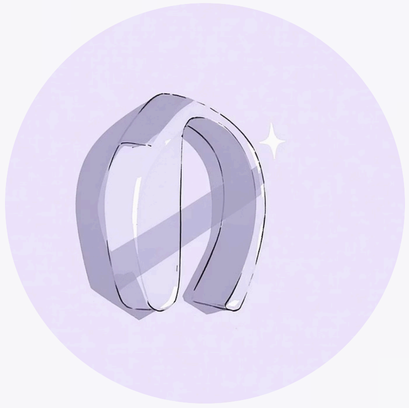
```
numbers = [1, 2, 3, 4, 5, 6]
doubled = [x * 2 for x in numbers]
evens = [x for x in numbers if x % 2 == 0]
squared_odds = [x**2 for x in numbers if x % 2 != 0]
```

These operations enable efficient data transformation, filtering, and manipulation, forming the backbone of many Python algorithms and data processing pipelines.

Tuples: Immutable Ordered Collections

Immutable by Design

Once created, tuple elements cannot be modified, added, or removed. This immutability provides data integrity and enables tuples to serve as dictionary keys.



Parentheses Syntax

Tuples are defined using parentheses: `('apple', 'banana', 'cherry')`. Single-element tuples require a trailing comma: `(42,)`

Fixed Data Collections

Perfect for representing data that shouldn't change: coordinates (x, y), RGB colors, database records, or configuration settings that remain constant.

Tuple Unpacking

Assign multiple variables simultaneously from tuple values, enabling elegant multiple returns from functions and efficient variable swapping.

```
# Multiple assignment
coordinates = (10, 20)
x, y = coordinates

# Function returns
def get_dimensions():
    return (1920, 1080)
width, height = get_dimensions()

# Variable swapping
a, b = 5, 10
a, b = b, a  # Swap values
```

Advantages of Tuples

- **Performance**: Faster than lists due to immutability
- **Safety**: Protect data from accidental modification
- **Hashable**: Can be used as dictionary keys
- **Memory efficient**: Consume less memory than lists
- **Semantic clarity**: Signal that data is fixed

When to use tuples: Use tuples for data that represents a single entity with multiple attributes (like a person's name and age), or when you want to ensure data immutability.

Dictionaries: Key-Value Mappings

Dictionaries are Python's implementation of associative arrays, providing fast lookups through key-value relationships. They're fundamental for modeling real-world entities and managing complex data relationships.

Dictionary Fundamentals

Dictionaries store data as pairs enclosed in curly braces, where each unique key maps to a specific value. This structure enables instant data retrieval without iteration.

Basic Structure

```
student = {
    'name': 'Alice Johnson',
    'age': 25,
    'grade': 'A',
    'courses': ['Python', 'Data Science']
}
```

Key Requirements

- Keys must be **immutable types**: strings, numbers, tuples
- Keys must be **unique**: duplicate keys overwrite previous values
- Keys are **case-sensitive**: 'Name' and 'name' are different

Value Flexibility

- Values can be **any data type**: strings, numbers, lists, other dictionaries
- Values can be **duplicated** across different keys
- Values can be **complex nested structures**

Accessing Dictionary Data

Direct Key Access

```
print(student['name'])     # Output: 'Alice Johnson'
print(student['age'])      # Output: 25
print(student['courses'])  # Output: ['Python', 'Data Science']
```

Safe Access with .get()

```
# Returns None if key doesn't exist
print(student.get('address'))  # Output: None

# Provide default value
print(student.get('address', 'Unknown'))  # Output: 'Unknown'
```

Modifying Dictionaries

```
# Adding new key-value pairs
student['email'] = 'alice@example.com'

# Updating existing values
student['age'] = 26

# Nested updates
student['courses'].append('Machine Learning')
```

KeyError Alert: Accessing a non-existent key with `dict[key]` raises a KeyError. Use `.get()` for safe access when keys might not exist.

Dictionary Methods & Usage

.keys()

Returns a view of all dictionary keys

```
student.keys()
```

.values()

Returns a view of all dictionary values

```
student.values()
```

.items()

Returns key-value pairs as tuples

```
student.items()
```

.get()

Safely retrieves values with default

```
student.get('age')
```

.pop()

Removes and returns value for key

```
student.pop('grade')
```

Practical Applications

Modeling Real-World Data

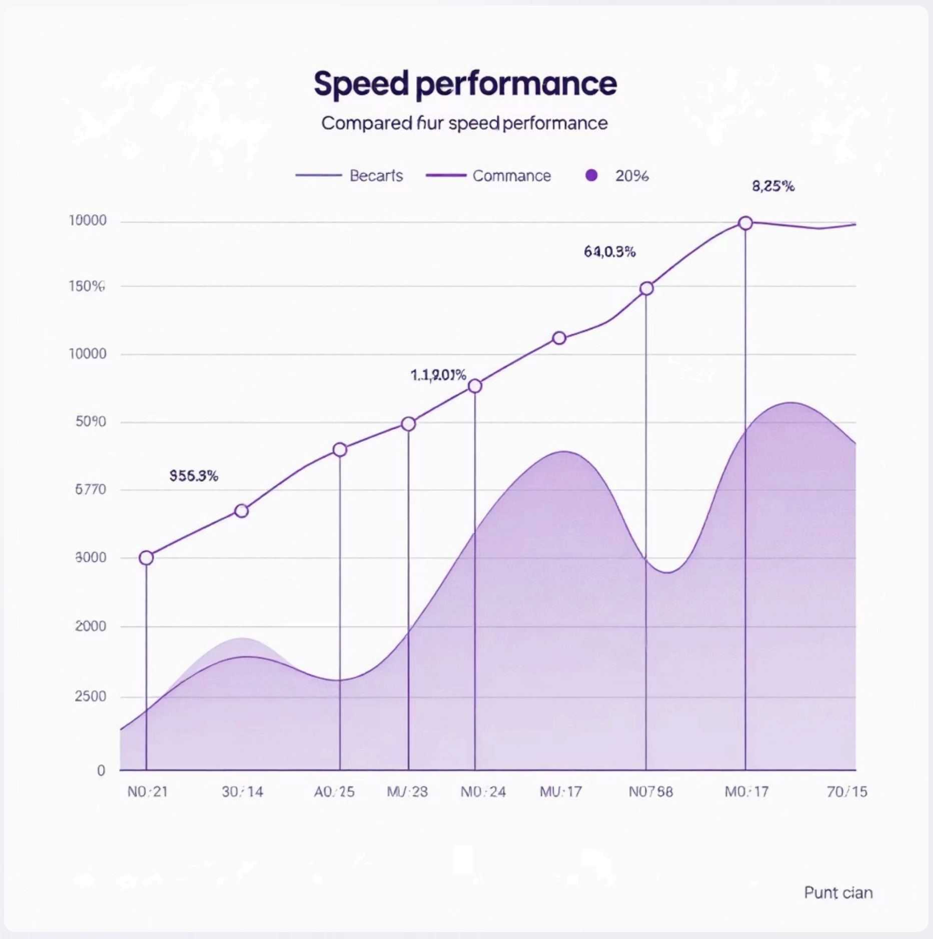
```
# Student Records System
students = {
    'S001': {'name': 'John', 'marks': 85},
    'S002': {'name': 'Emma', 'marks': 92},
    'S003': {'name': 'Alex', 'marks': 78}
}

# Inventory Management
inventory = {
    'laptop': {'price': 45000, 'stock': 15},
    'mouse': {'price': 500, 'stock': 50},
    'keyboard': {'price': 1200, 'stock': 30}
}
```

Dictionary Comprehensions

Create dictionaries efficiently using comprehension syntax:

```
# Square numbers
squares = {x: x**2 for x in range(1, 6)}
# Output: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

# Filter and transform
prices = {'apple': 50, 'banana': 30, 'cherry': 80}
expensive = {k: v for k, v in prices.items() if v > 40}
# Output: {'apple': 50, 'cherry': 80}
```

Performance Advantages



Dictionaries provide **O(1) average-case lookup time**, making them significantly faster than lists for searching operations.

Comparison: Dict vs List Search

- **Dictionary lookup**: Instant retrieval regardless of size
- **List search**: Must check each element sequentially
- For 1 million items: dict is ~1000x faster

💡 **Best Practice**

Use dictionaries when you need frequent lookups by identifier. Use lists when order matters more than lookup speed, or when you need numeric indexing.

Combining Data Structures in Python

Real-world applications rarely use isolated data structures. Combining lists, tuples, and dictionaries creates powerful, flexible data models that represent complex relationships efficiently.

Lists of Dictionaries

Store multiple records where each entry has named attributes

Nested Structures

Create hierarchical data with dictionaries containing lists or other dictionaries

Tuples in Collections

Use tuples for immutable groupings within mutable containers

Practical Example: Student Management System

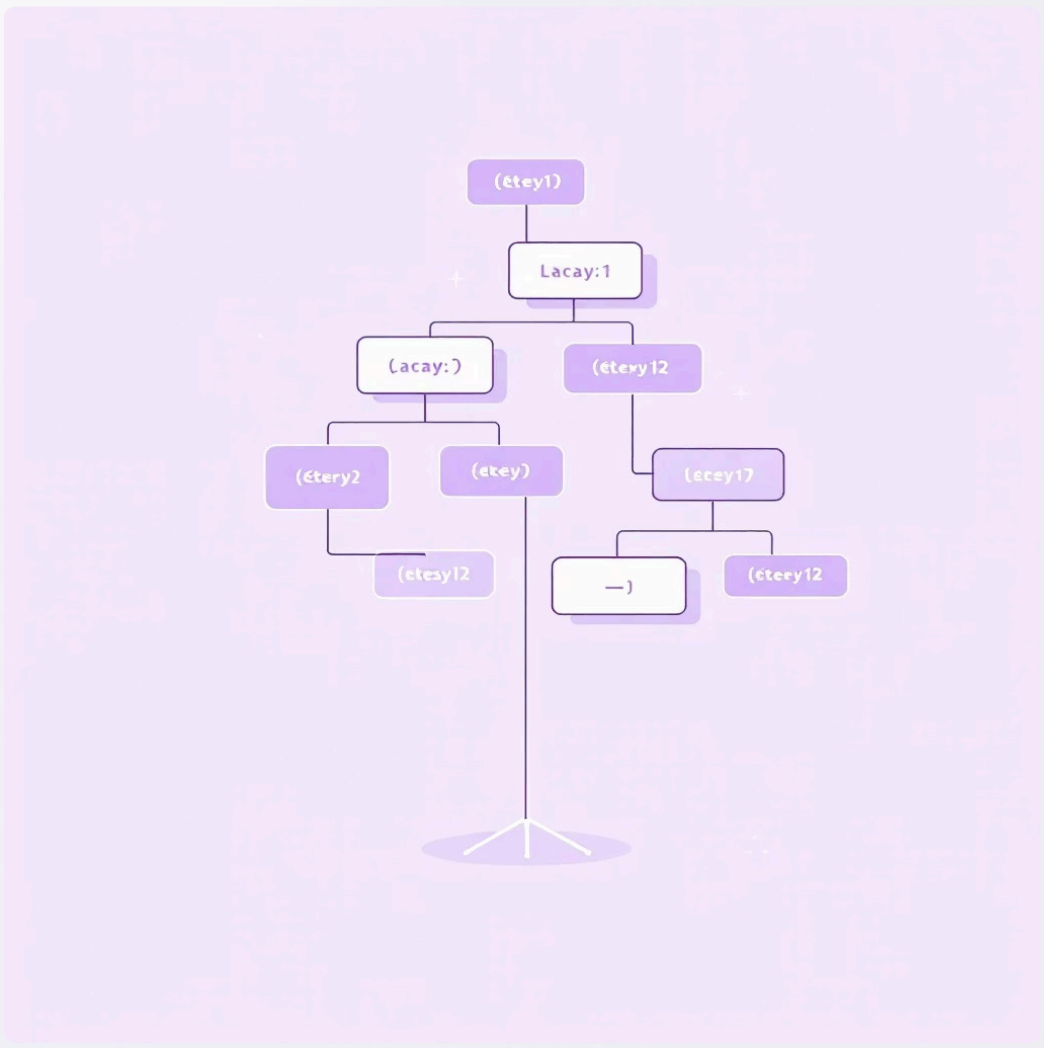
```
# List of dictionaries for student records
students = [
    {
        'id': 'VTU001',
        'name': 'Priya Sharma',
        'age': 20,
        'grades': {'Python': 95, 'DSA': 88, 'DBMS': 92},
        'contact': ('priya@example.com', '9876543210')
    },
    {
        'id': 'VTU002',
        'name': 'Rahul Kumar',
        'age': 21,
        'grades': {'Python': 87, 'DSA': 91, 'DBMS': 85},
        'contact': ('rahul@example.com', '9845678901')
    },
    {
        'id': 'VTU003',
        'name': 'Ananya Reddy',
        'age': 19,
        'grades': {'Python': 93, 'DSA': 89, 'DBMS': 94},
        'contact': ('ananya@example.com', '9823456789')
    }
]

# Accessing nested data
print(students[0]['name'])          # 'Priya Sharma'
print(students[1]['grades']['DSA'])  # 91
email, phone = students[2]['contact']
print(email)                        # 'ananya@example.com'
```

Benefits of Combined Structures

- **Rich data modeling**: Represent entities with multiple attributes and relationships
- **Flexibility**: Mix mutable and immutable components as needed
- **Scalability**: Easily add records or attributes without restructuring
- **Real-world mapping**: Mirrors how data exists in databases and APIs

Common Patterns

- **Configuration data**: Dict with tuples for server settings
- **Database rows**: List of dicts representing table records
- **Coordinate systems**: List of tuples for (x, y) points
- **JSON-like structures**: Nested dicts and lists for API data



Algorithm Design Benefits

Leveraging appropriate data structure combinations leads to cleaner, more maintainable code with better performance characteristics. Understanding when to nest structures enables efficient solutions to complex problems while maintaining code readability.

Summary & Next Steps

1

Master the Fundamentals

Achieving mastery of lists, tuples, and dictionaries is absolutely essential for Python programming success. These structures form the foundation upon which all Python applications are built.

2

Practice Daily

Dedicate time each day to creating, accessing, and manipulating these structures through hands-on coding exercises, LeetCode problems, and mini-projects.

3

Explore Advanced Topics

Expand your knowledge into sets for unique collections, the collections module (Counter, defaultdict, OrderedDict), and designing custom data structures for specialized needs.

Key Takeaways

- **Lists**: Use for ordered, mutable collections that change frequently
- **Tuples**: Choose for immutable, fixed data and function returns
- **Dictionaries**: Ideal for key-value mappings and fast lookups
- Combine structures for powerful, flexible data models
- Select structures based on mutability, ordering, and performance needs

Practice Exercises

1. Build a contact management system using dictionaries
2. Implement sorting algorithms on lists
3. Create a grade calculator combining all three structures
4. Parse and manipulate JSON-like nested data

Learning Resources

Official Documentation

Python.org comprehensive guides

W3Schools Python

Interactive tutorials and examples

VTU Syllabus

University reference materials

Coding Practice

HackerRank, LeetCode challenges

Exam Preparation Tip: Focus on understanding when to use each structure and practice writing code by hand. VTU exams often test practical implementation and structure selection reasoning.

Continue building on this foundation by tackling progressively complex programming challenges. The structures covered in Module 2 will appear in every Python program you write, making this knowledge invaluable for your academic and professional journey.