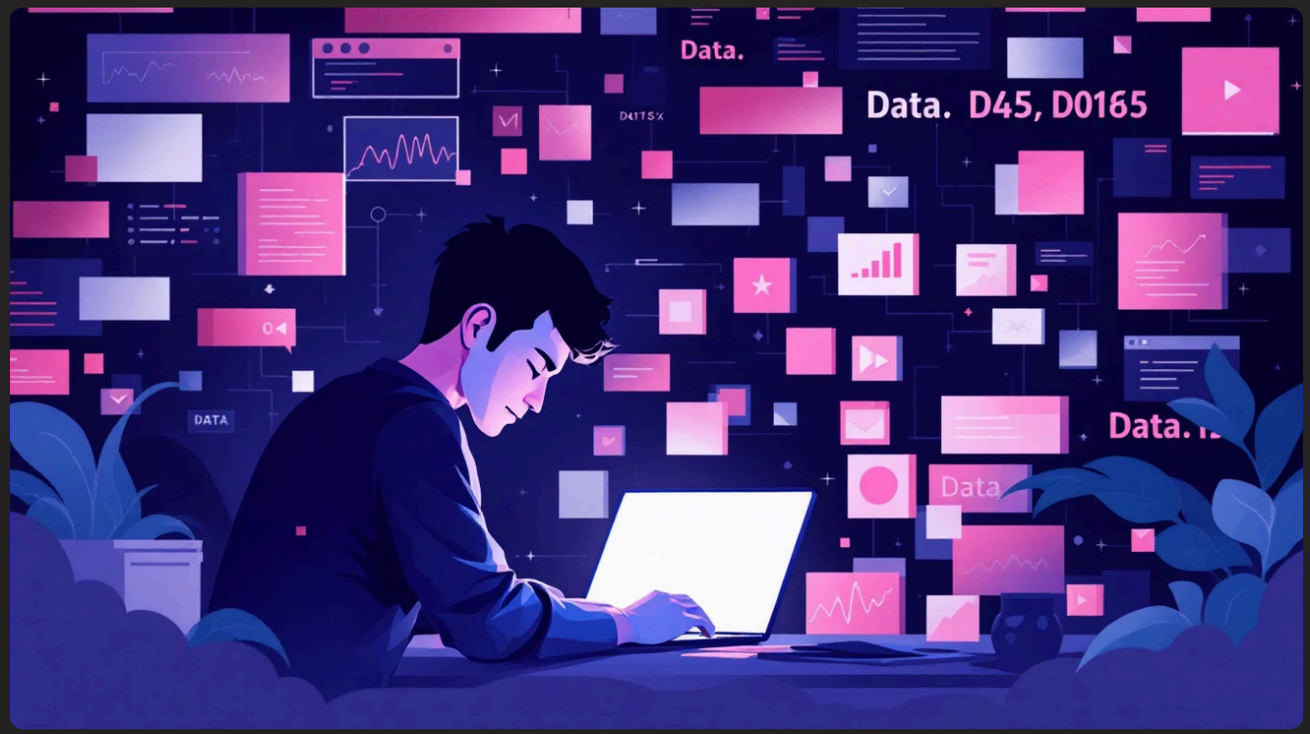
Python Data Structures: Lists, Tuples, Sets, and Dictionaries

Master the essential building blocks for efficient Python programming

FUNDAMENTALS

Why Python Data Structures Matter



Data structures are the foundation of every Python program. They determine how you organize, access, and manipulate information efficiently.

Choosing the right structure directly impacts your code's performance, readability, and maintainability. Understanding when to use each type is essential for writing clean, professional Python code.

Lists

Ordered, mutable sequences for dynamic collections

Tuples

Immutable sequences for fixed data and safe storage

Sets

Unordered collections for unique elements

Dictionaries

Key-value pairs for fast lookups

LISTS

Python Lists: Ordered, Mutable Collections

Key Characteristics

- Ordered: Elements maintain insertion order
- Mutable: Can be modified after creation
- Allows duplicates and mixed data types
- Dynamic sizing - grows or shrinks automatically

Common Applications

- Storing user inputs or query results
- Managing shopping cart items
- Processing sequences of data
- Implementing stacks and queues

Syntax & Examples

```
# Creating lists
my_list = [1, 2, 3, 'apple', 5.5]
empty_list = []

# Common operations
my_list.append('banana') # Add item
my_list.remove(2) # Remove item
my_list.sort() # Sort list
item = my_list[0] # Access by index

# List comprehension
squares = [x**2 for x in range(5)]
# Result: [0, 1, 4, 9, 16]
```

TUPLES

Python Tuples: Immutable Sequences

Key Characteristics

- Ordered: Elements maintain insertion order
- Immutable: Cannot be modified after creation
- Allows duplicates and mixed data types
- Fixed size: Size is determined at creation
- Hashable: Can be used as dictionary keys

Common Applications

- Function return values (e.g., multiple values)
- Dictionary keys (due to immutability)
- Protecting data that should not change
- Unpacking values into variables
- Representing fixed collections like coordinates

Syntax & Examples

```
# Creating tuples
my_tuple = (1, 'banana', 3)
single_item = (4,) # Comma is crucial
empty_tuple = ()

# Accessing elements
item = my_tuple[1] # 'banana'

# Unpacking tuples
a, b, c = my_tuple
# a='1', b='banana', c='3'

# Tuple operations (creates new tuple)
new_tuple = my_tuple + (4, 5)
sliced_tuple = my_tuple[1:]

# Tuples as dictionary keys
coords = {(0, 0): 'origin', (1, 1): 'point'}
```

SETS

Python Sets: Unordered Collections of Unique Elements

Key Characteristics

- Unordered: Elements do not maintain insertion order
- Mutable: Can be modified after creation (add/remove elements)
- Unique elements only: No duplicate values allowed
- Fast membership testing (ìn keyword)
- Elements must be hashable (immutable types like numbers, strings, tuples)

Common Applications

- Removing duplicates from a sequence
- Efficient membership testing
- Mathematical set operations (union, intersection, difference)
- Finding unique values across multiple collections
- Deduplication of data

Syntax & Examples

```
# Creating sets
my_set = {1, 2, 3, 'apple', 3} # Duplicates ignored
# Result: {1, 2, 3, 'apple'}
empty_set = set() # Use set() for an empty set

# Adding and removing elements
my_set.add('banana')
my_set.remove(2) # Raises KeyError if element not found
my_set.discard(4) # Does nothing if element not found

# Set operations
set_a = {1, 2, 3, 4}
set_b = {3, 4, 5, 6}
union_set = set_a | set_b # {1, 2, 3, 4, 5, 6}
intersection_set = set_a & set_b # {3, 4}
difference_set = set_a - set_b # {1, 2}

# Membership testing
is_present = 'apple' in my_set # True
```

DICTIONARIES

Python Dictionaries: Key-Value Pairs for Fast Lookups

Key Characteristics

- Unordered: Elements do not maintain insertion order (conceptually)
- Mutable: Can be modified (add, remove, change items)
- Stores data as key: value pairs
- Keys must be unique and hashable (e.g., numbers, strings, tuples)
- Values can be any data type and can be duplicates
- Fast average-case time complexity (O(1)) for lookups

Common Applications

- Storing user profiles and object properties
- Caching data for quick retrieval
- Configuration settings in applications
- Mapping relationships between data elements
- Representing JSON-like data structures

Syntax & Examples

```
# Creating dictionaries
my_dict = {"name": "Alice", "age": 30}
empty_dict = {}

# Accessing values
name = my_dict["name"] # "Alice"
age = my_dict.get("age") # 30 (safer than direct access)

# Adding/Updating items
my_dict["city"] = "New York" # Add new
my_dict["age"] = 31 # Update existing

# Removing items
del my_dict["city"]
# or my_dict.pop("age")

# Iterating through dictionaries
for key, value in my_dict.items():
    # print(key, value)
    pass
```

AI APPLICATIONS

Python Data Structures in Generative AI

Leveraging the right data structure is crucial for optimizing performance and managing complex data workflows in AI/ML development.

Lists

Ideal for storing sequences of training data, batches of samples, token sequences in NLP, and capturing dynamic model outputs.

Tuples

Used for immutable model parameters, representing coordinate pairs in image processing, and returning multiple fixed values from functions.

Sets

Essential for removing duplicate training samples, tracking unique tokens in a vocabulary, and efficient dataset deduplication for performance.

Dictionaries

Perfect for storing model configurations, mapping token IDs to embeddings, caching inference results, and managing hyperparameters.