Core Data Structures in Python

Lists, Implementation, and Hash Tables

Martin Benning

University College London

ENGF0034 – Design and Professional Skills

Lecture Overview

Objectives

To understand fundamental Python data structures (Lists, Dictionaries, Sets), explore the Python memory model, delve into the underlying implementations (Dynamic Arrays and Hash Tables), and analyse their performance characteristics (Complexity and Amortization).

1. Python Lists: Interface and Usage
2. The Python Memory Model and References
3. List Implementation: Dynamic Arrays
4. Performance Analysis and Amortisation
5. Hash Tables: Dictionaries and Sets
6. Conclusion

Python Lists: Interface and Usage

Data Structure of the Week: The List

Motivation

Programming frequently requires storing and managing collections of data. The Python `list` is a primary built-in structure for this.

Data Structure of the Week: The List

Motivation

Programming frequently requires storing and managing collections of data. The Python `list` is a primary built-in structure for this.

Definition

A Python list is a **mutable**, **ordered sequence** of elements.

Data Structure of the Week: The List

Motivation

Programming frequently requires storing and managing collections of data. The Python `list` is a primary built-in structure for this.

Definition

A Python list is a **mutable**, **ordered sequence** of elements.

- **Ordered:** Elements maintain a defined sequence, accessible via index.

Data Structure of the Week: The List

Motivation

Programming frequently requires storing and managing collections of data. The Python `list` is a primary built-in structure for this.

Definition

A Python list is a **mutable**, **ordered sequence** of elements.

- **Ordered:** Elements maintain a defined sequence, accessible via index.
- **Mutable:** The list can be modified in place after creation.

Data Structure of the Week: The List

Motivation

Programming frequently requires storing and managing collections of data. The Python `list` is a primary built-in structure for this.

Definition

A Python list is a **mutable**, **ordered sequence** of elements.

- **Ordered:** Elements maintain a defined sequence, accessible via index.
- **Mutable:** The list can be modified in place after creation.
- **Dynamic:** The size can change (grow and shrink).

Creating and Accessing Lists

Syntax and Indexing

Lists are created using square brackets []. Access is via zero-based indexing.

```python
# Creating a list
days = ["mon", "tues", "weds", "thurs", "fri"]

# Checking the length
count = len(days)  # 5

# Accessing elements (Zero-based index)
day2 = days[1]    # "tues"

# Negative indexing (from the end)
last_day = days[-1] # "fri"
```

Modifying Lists (Mutability)

Lists can be changed in place.

Item Assignment

```
days = ["mon", "tues", "weds", "thurs", "fri"]
days[4] = "FRIDAY!!!"
# days is now ["mon", "tues", "weds", "thurs", "FRIDAY!!!"]
```

Modifying Lists (Mutability)

Item Assignment

```
days = ["mon", "tues", "weds", "thurs", "fri"]
days[4] = "FRIDAY!!!"
# days is now ["mon", "tues", "weds", "thurs", "FRIDAY!!!"]
```

Adding Elements

- `.append(element)`: Adds to the end.
- `.insert(index, element)`: Inserts at a specific index.

```
days.append("sat")
# Insert "sun" at the beginning
days.insert(0, "sun")
```

Removing Elements

Methods for Removal

- `del list[i]`: Deletes the element at index `i`.
- `.pop()`: Removes and returns the last element.
- `.pop(i)`: Removes and returns the element at index `i`.

```python
days = ["mon", "tues", "weds", "thurs", "fri"]

# I don't like mondays
del days[0]
# days is now ["tues", "weds", "thurs", "fri"]

# Remove the last element
last = days.pop() # last is "fri"
```

Iterating Over Lists I

Iteration using Index (Less Pythonic)

Traditional approach using `range(len(...))`.

```
# iteration over a list using an index
days = ["mon", "tues", "weds"]
for day_num in range(0, len(days)):
    print(days[day_num])
```

Iterating Over Lists I

Iteration using Index (Less Pythonic)

Traditional approach using `range(len(...))`.

```
# iteration over a list using an index
days = ["mon", "tues", "weds"]
for day_num in range(0, len(days)):
    print(days[day_num])
```

Native Iteration (Pythonic)

The preferred method. Directly iterates over the elements.

```
# native iteration over a list (more pythonic!)
for day in days:
    print(day)
```

Iterating Over Lists I

Iteration using Index (Less Pythonic)

Traditional approach using `range(len(...))`.

```python
# iteration over a list using an index
days = ["mon", "tues", "weds"]
for day_num in range(0, len(days)):
    print(days[day_num])
```

Native Iteration (Pythonic)

The preferred method. Directly iterates over the elements.

```python
# native iteration over a list (more pythonic!)
for day in days:
    print(day)
```

Iterating Over Lists II

Native Iteration (Pythonic)

The preferred method. Directly iterates over the elements.

```python
# native iteration over a list (more pythonic!)
for day in days:
    print(day)
```

Using `enumerate`

If both the index and the item are needed.

```python
for i, day in enumerate(days):
    print(f"Index {i}: {day}")
```

The Python Memory Model and References

Understanding Variables in Python

To understand how lists behave, especially when copied or passed to functions, we must understand the Python memory model.

The "Variables as Boxes" Metaphor (Incorrect for Python)

In languages like C, a variable is often thought of as a memory location that holds a value.

Understanding Variables in Python

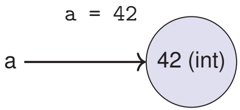
To understand how lists behave, especially when copied or passed to functions, we must understand the Python memory model.

The "Variables as Boxes" Metaphor (Incorrect for Python)

In languages like C, a variable is often thought of as a memory location that holds a value.

The "Variables as Labels" Metaphor (Correct)

In Python, variables are **names** (labels) that **refer** to objects in memory. Assignment (=) binds a name to an object.



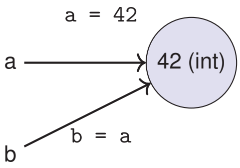
Understanding Variables in Python

The ”Variables as Boxes” Metaphor (Incorrect for Python)

In languages like C, a variable is often thought of as a memory location that holds a value.

The ”Variables as Labels” Metaphor (Correct)

In Python, variables are **names** (labels) that **refer** to objects in memory. Assignment (=) binds a name to an object.

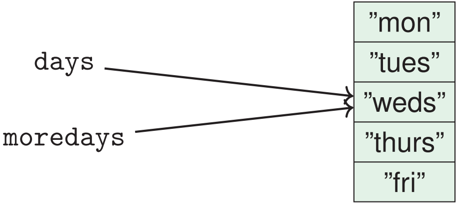


The Impact of References: Aliasing

When multiple names refer to the same mutable object (like a list), this is called **aliasing**.

```
# 1. Create a list object and bind 'days' to it
days = ["mon", "tues", "weds", "thurs", "fri"]

# 2. Bind the name 'moredays' to the SAME object.
# This does NOT copy the list!
moredays = days
```



The Consequences of Aliasing

Because the object is mutable, changes made through one alias are visible through all other aliases.

```python
days = ["mon", "tues", "weds", "thurs", "FRIDAY!!!"]
moredays = days

# Modify the list via the 'moredays' alias
moredays[4] = "fri"

# Check the original 'days' variable
print(days)
# Output: ["mon", "tues", "weds", "thurs", "fri"]
```

The Consequences of Aliasing

```
days = ["mon", "tues", "weds", "thurs", "FRIDAY!!!"]
moredays = days

# Modify the list via the 'moredays' alias
moredays[4] = "fri"

# Check the original 'days' variable
print(days)
# Output: ["mon", "tues", "weds", "thurs", "fri"]
```

A Common Source of Bugs

This behaviour is often surprising and is a major source of errors, especially when passing mutable arguments to functions.

How to Copy a List I

If we want an independent copy, we must explicitly create a new list object.

Methods for Shallow Copying

```python
days = ["mon", "tues", "weds", "thurs", "fri"]

# 1. Using the .copy() method (Recommended)
copy1 = days.copy()

# 2. Using a full slice (Idiomatic Python)
copy2 = days[:]
```

How to Copy a List I

Methods for Shallow Copying

```python
days = ["mon", "tues", "weds", "thurs", "fri"]

# 1. Using the .copy() method (Recommended)
copy1 = days.copy()

# 2. Using a full slice (Idiomatic Python)
copy2 = days[:]
```

Verification

```python
copy1[0] = "YAWN!"
# copy1 is ["YAWN!", "tues", ...]
# days is still ["mon", "tues", ...]
```

How to Copy a List I

Methods for Shallow Copying

```python
days = ["mon", "tues", "weds", "thurs", "fri"]

# 1. Using the .copy() method (Recommended)
copy1 = days.copy()

# 2. Using a full slice (Idiomatic Python)
copy2 = days[:]
```

Verification

```python
copy1[0] = "YAWN!"
# copy1 is ["YAWN!", "tues", ...]
# days is still ["mon", "tues", ...]
```

How to Copy a List II

If we want an independent copy, we must explicitly create a new list object.

Verification

```
copy1[0] = "YAWN!"
# copy1 is ["YAWN!", "tues", ...]
# days is still ["mon", "tues", ...]
```

Shallow vs. Deep Copy

`.copy()` creates a shallow copy. If the list contains nested mutable objects (e.g., lists within lists), those inner objects are still shared. Use `copy.deepcopy()` if needed for full independence.

Shallow Copy: The Pitfall

Example with Nested Lists

```python
list_a = [[1, 2], [3, 4]]
# Create a shallow copy
list_b = list_a.copy()
# Modify an inner list via list_b
list_b[0][0] = 99
# The change is reflected in list_a!
print(list_a) # Output: [[99, 2], [3, 4]]
```

Shallow Copy: The Pitfall

Example with Nested Lists

```python
list_a = [[1, 2], [3, 4]]
# Create a shallow copy
list_b = list_a.copy()
# Modify an inner list via list_b
list_b[0][0] = 99
# The change is reflected in list_a!
print(list_a) # Output: [[99, 2], [3, 4]]
```

Why This Happens

The `.copy()` method created a new outer list for `list_b`, but it filled it with pointers to the **same inner lists** that `list_a` was pointing to.

List Implementation: Dynamic Arrays

Looking Under the Hood

We know the interface (how to use lists). To understand *why* certain operations are fast and others slow (performance characteristics), we must examine the implementation.

The Underlying Data Structure

Python lists are implemented as **Dynamic Arrays** (similar to C++ `std::vector` or Java `ArrayList`).

Looking Under the Hood

We know the interface (how to use lists). To understand *why* certain operations are fast and others slow (performance characteristics), we must examine the implementation.

The Underlying Data Structure

Python lists are implemented as **Dynamic Arrays** (similar to C++ `std::vector` or Java `ArrayList`).

Key Feature: Contiguous Memory

A dynamic array uses a contiguous block of memory to store its data. This is crucial for performance.

Looking Under the Hood

The Underlying Data Structure

Python lists are implemented as **Dynamic Arrays** (similar to C++ `std::vector` or Java `ArrayList`).

Key Feature: Contiguous Memory

A dynamic array uses a contiguous block of memory to store its data. This is crucial for performance.

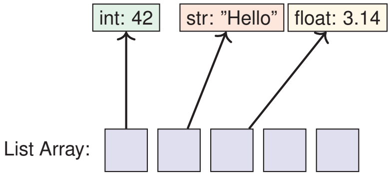
The Heterogeneous Challenge

Python lists can store different types, which have different sizes (e.g., an integer vs. a long string). How can they be stored contiguously?

Implementation: Array of Pointers

The Solution

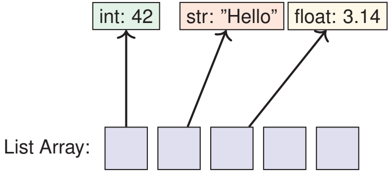
The list does not store the actual objects directly. Instead, it stores an array of **pointers** (references) to the objects.



Implementation: Array of Pointers

The Solution

The list does not store the actual objects directly. Instead, it stores an array of **pointers** (references) to the objects.



Implications

All pointers have the same size (e.g., 64 bits), regardless of the object they point to. This maintains contiguity and enables fast indexing.

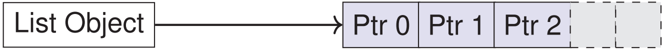
The CPython List Structure

In CPython (the standard implementation), a list object has metadata alongside the pointer array.

Conceptual Structure (Simplified)

A list object contains:

1. `ob_size`: The current number of elements (the length, `len(L)`).
2. `allocated`: The total capacity of the underlying array.
3. `ob_item`: A pointer to the dynamic array of pointers.



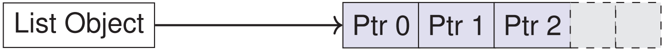
Size: 3, Allocated: 5

The CPython List Structure

Conceptual Structure (Simplified)

A list object contains:

1. `ob_size`: The current number of elements (the length, `len(L)`).
2. `allocated`: The total capacity of the underlying array.
3. `ob_item`: A pointer to the dynamic array of pointers.



Size: 3, Allocated: 5

Key Insight: Over-allocation

Crucially, `allocated` is often greater than `ob_size`. This strategy is called **Over-allocation**.

Performance Analysis and Amortisation

Analysing Performance (Big O)

We use Big O notation to describe how the time required scales with the length of the list (N).

Constant Time Operations: $\mathcal{O}(1)$

- **Indexing (`L[i]`):** Due to contiguous memory, the address of the i-th pointer can be calculated directly: `Base + i * PointerSize`.
- **Length (`len(L)`):** The size is stored in the metadata (`ob_size`) and retrieved instantly.

Analysing Performance (Big O)

We use Big O notation to describe how the time required scales with the length of the list (N).

Constant Time Operations: $\mathcal{O}(1)$

- **Indexing (`L[i]`):** Due to contiguous memory, the address of the i-th pointer can be calculated directly: `Base + i * PointerSize`.
- **Length (`len(L)`):** The size is stored in the metadata (`ob_size`) and retrieved instantly.

Empirical Evidence

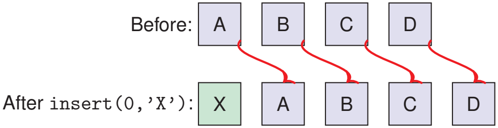
Experiments (e.g., using the logic in `testaccess.py`) confirm that access time is independent of the list size.

The Cost of Insertion and Deletion

The dynamic array implementation involves a trade-off when modifying the structure.

The Problem: Maintaining Contiguity

If we insert or delete an element in the middle, all subsequent elements must be shifted to maintain order and prevent gaps.

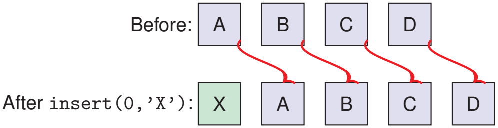


The Cost of Insertion and Deletion

The dynamic array implementation involves a trade-off when modifying the structure.

The Problem: Maintaining Contiguity

If we insert or delete an element in the middle, all subsequent elements must be shifted to maintain order and prevent gaps.



Computational Cost

Shifting N items requires N memory operations. This is $\mathcal{O}(N)$.

Complexity of Insertion/Deletion I

Worst Case: Beginning of the List

`L.insert(0, x)` or `L.pop(0)`.

- Requires shifting all N elements.
- **Complexity:** $\mathcal{O}(N)$ **(Linear Time)**

Complexity of Insertion/Deletion I

Worst Case: Beginning of the List

`L.insert(0, x)` or `L.pop(0)`.

- Requires shifting all N elements.
- **Complexity:** $\mathcal{O}(\mathrm{N})$ **(Linear Time)**

Average Case: Middle of the List

`L.insert(i, x)` or `L.pop(i)`.

- Requires shifting $\mathrm{N} - \mathrm{i}$ elements (on average N/2).
- **Complexity:** $\mathcal{O}(\mathrm{N})$ **(Linear Time)**

Complexity of Insertion/Deletion II

Best Case: End of the List

`L.pop()`.

- No shifting required.
- **Complexity:** $\mathcal{O}(1)$ **(Constant Time)**

Complexity of Insertion/Deletion II

Best Case: End of the List

`L.pop()`.

- No shifting required.
- **Complexity:** $\mathcal{O}(1)$ **(Constant Time)**

Warning

Using `pop(0)` repeatedly (e.g., using a list as a queue) leads to $\mathcal{O}(N^2)$ total time. Use `collections.deque` for efficient queues.

The Complexity of Append

What about `L.append(x)`?

The Fast Path (Thanks to Over-allocation)

If `Size < Allocated`, there is free space.

1. Place the new pointer in the next slot.
2. Increment the Size.

This is clearly $\mathcal{O}(1)$.

The Complexity of Append

The Fast Path (Thanks to Over-allocation)

If `Size < Allocated`, there is free space.

1. Place the new pointer in the next slot.
2. Increment the Size.

This is clearly $\mathcal{O}(1)$.

The Slow Path: Resizing

If `Size == Allocated` (The list is full).

1. Allocate a new, larger array.
2. Copy all N existing pointers to the new array.
3. Free the old array.

The Complexity of Append

The Slow Path: Resizing

If `Size == Allocated` (The list is full).

1. Allocate a new, larger array.
2. Copy all N existing pointers to the new array.
3. Free the old array.

This resize operation takes $\mathcal{O}(N)$ time.

The Dilemma

The worst-case complexity of a single `append()` is $\mathcal{O}(N)$. How can we claim it is efficient?

The Importance of the Growth Strategy

How much extra space should be allocated during a resize?

Naive Strategy: Additive Growth (+C)

Allocate a constant amount of extra space (e.g., +1 slot).

- Problem: Resizing happens too frequently.
- Total time for N appends becomes $\mathcal{O}(N^2)$. (The sum of an arithmetic series).

The Importance of the Growth Strategy

How much extra space should be allocated during a resize?

Naive Strategy: Additive Growth (+C)

Allocate a constant amount of extra space (e.g., +1 slot).

- Problem: Resizing happens too frequently.
- Total time for N appends becomes $\mathcal{O}(N^2)$. (The sum of an arithmetic series).

Correct Strategy: Geometric Growth ($\times$ Factor)

Multiply current size by a constant factor (e.g., double the size). This is what Python does.

- Advantage: Resizing happens exponentially less frequently as the list grows.

The Importance of the Growth Strategy

Naive Strategy: Additive Growth (+C)

Allocate a constant amount of extra space (e.g., +1 slot).

- Problem: Resizing happens too frequently.
- Total time for N appends becomes $\mathcal{O}(N^2)$. (The sum of an arithmetic series).

Correct Strategy: Geometric Growth ($\times$ Factor)

Multiply current size by a constant factor (e.g., double the size). This is what Python does.

- Advantage: Resizing happens exponentially less frequently as the list grows.

CPython's Growth Factor

CPython uses a growth factor of approximately 1.125 (9/8).

Amortised Analysis I

When an operation is usually fast but occasionally slow, we use **amortised analysis** to find the average time taken per operation over a sequence of operations.

Analysis of Geometric Growth

The cost of the expensive $\mathcal{O}(N)$ resize is "amortised" (spread out) over the many cheap $\mathcal{O}(1)$ appends that preceded it.

Consider N appends using the doubling strategy.

- The total cost of copying occurs at sizes $1, g, g^2, \ldots, g^k$ where g is the growth factor and $N \approx g^{k+1}$.
- The total number of copies is a geometric series, i.e.

$$C(N) = \sum_{i=0}^{k} g^i = \frac{g^{k+1} - 1}{g - 1} \approx \frac{N}{g - 1}$$

Amortised Analysis I

Analysis of Geometric Growth

- The total cost of copying occurs at sizes $1, g, g^2, \dots, g^k$ where g is the growth factor and $N \approx g^{k+1}$.
- The total number of copies is a geometric series, i.e.

$$C(N) = \sum_{i=0}^{k} g^i = \frac{g^{k+1} - 1}{g - 1} \approx \frac{N}{g - 1}$$

- For a doubling strategy ($g = 2$), the total copy cost is $\approx \frac{N}{2-1} = N$.
- For CPython's strategy ($g \approx 1.125$), the cost is $\approx \frac{N}{0.125} = 8N$.
- In both cases, the total copy cost is linear, i.e., $\mathcal{O}(N)$.

Amortised Analysis II

When an operation is usually fast but occasionally slow, we use **amortised analysis** to find the average time taken per operation over a sequence of operations.

Total Cost

Total cost for N operations = Cost of inserts ($\approx N$) + Cost of resizing ($\approx 2N$) = $3N$.

Amortized Complexity

The total time for N appends is $\mathcal{O}(N)$. The average (amortized) cost per append is $\mathcal{O}(N)/N = \mathcal{O}(1)$.

List Performance Summary

Operation	**Example**	**Average Complexity**
Indexing / Assignment	`L[i]`	$\mathcal{O}(1)$
Length	`len(L)`	$\mathcal{O}(1)$
Append (End)	`L.append(x)`	$\mathcal{O}(1)$ (Amortised)
Pop (End)	`L.pop()`	$\mathcal{O}(1)$
Insert (Start/Middle)	`L.insert(0, x)`	$\mathcal{O}(N)$
Pop (Start/Middle)	`L.pop(0)`	$\mathcal{O}(N)$
Search (Membership)	`x in L`	$\mathcal{O}(N)$
Sort	`L.sort()`	$\mathcal{O}(N \log N)$

List Performance Summary

Operation	Example	Average Complexity
Indexing / Assignment	`L[i]`	$\mathcal{O}(1)$
Length	`len(L)`	$\mathcal{O}(1)$
Append (End)	`L.append(x)`	$\mathcal{O}(1)$ (Amortised)
Pop (End)	`L.pop()`	$\mathcal{O}(1)$
Insert (Start/Middle)	`L.insert(0, x)`	$\mathcal{O}(N)$
Pop (Start/Middle)	`L.pop(0)`	$\mathcal{O}(N)$
Search (Membership)	`x in L`	$\mathcal{O}(N)$
Sort	`L.sort()`	$\mathcal{O}(N \log N)$

The Limitation

Lists are excellent for ordered sequences, but searching for a value (`x in L`) requires a slow linear scan ($\mathcal{O}(N)$).

Hash Tables: Dictionaries and Sets

The Search Problem

Motivation

Searching a list takes $\mathcal{O}(N)$ time. This is too slow if we perform many lookups on large datasets (e.g., checking if a username exists).

The Search Problem

Motivation

Searching a list takes $\mathcal{O}(N)$ time. This is too slow if we perform many lookups on large datasets (e.g., checking if a username exists).

The Goal: Constant Time Lookup

Can we design a data structure that allows lookups in $\mathcal{O}(1)$ time, regardless of the collection size? Yes, using **Hash Tables**.

The Search Problem

Motivation

Searching a list takes $\mathcal{O}(N)$ time. This is too slow if we perform many lookups on large datasets (e.g., checking if a username exists).

The Goal: Constant Time Lookup

Can we design a data structure that allows lookups in $\mathcal{O}(1)$ time, regardless of the collection size? Yes, using **Hash Tables**.

Python Implementations

Hash tables are the foundation for Python's `dict` (Dictionary) and `set` data types.

Hash Functions

Definition

A hash function is a deterministic function that maps an input value (e.g., a string or number) to a fixed-size integer (the hash).

Input Key (e.g., ”Alice”) → Hash Function → Hash Value (Integer)

Hash Functions

Definition

A hash function is a deterministic function that maps an input value (e.g., a string or number) to a fixed-size integer (the hash).

Input Key (e.g., ”Alice”) $\longrightarrow$ Hash Function $\longrightarrow$ Hash Value (Integer)

Properties

- **Deterministic:** Same input always yields the same output.
- **Efficient:** Fast to compute ($\mathcal{O}(1)$).
- **Uniform Distribution:** Spreads inputs evenly to minimise collisions (different inputs mapping to the same hash).

The Hash Table Concept

A hash table uses the hash function to determine where to store data in an underlying array (the ”buckets”).

Conceptual Implementation

1. Start with an array of size M.

The Hash Table Concept

A hash table uses the hash function to determine where to store data in an underlying array (the ”buckets”).

Conceptual Implementation

1. Start with an array of size M.
2. To insert an item `K`:
   - Compute its hash: $\mathrm{H} = \text{hash}(\mathrm{K})$.
   - Determine the array index: $\text{Index} = \mathrm{H} \pmod{\mathrm{M}}$.
   - Store `K` at that index.

The Hash Table Concept

A hash table uses the hash function to determine where to store data in an underlying array (the ”buckets”).

Conceptual Implementation

1. Start with an array of size M.
2. To insert an item `K`:
   - Compute its hash: $H = \text{hash}(K)$.
   - Determine the array index: Index $= H \pmod M$.
   - Store `K` at that index.
3. To look up an item `K`:
   - Compute the hash and index.
   - Check if `K` is stored at that location.

The Hash Table Concept

Conceptual Implementation

1. Start with an array of size M.
2. To insert an item $\mathtt{K}$:
   - Compute its hash: $H = \text{hash}(K)$.
   - Determine the array index: $\text{Index} = H \pmod{M}$.
   - Store $\mathtt{K}$ at that index.
3. To look up an item $\mathtt{K}$:
   - Compute the hash and index.
   - Check if $\mathtt{K}$ is stored at that location.

Performance

If the hash function is fast and collisions are rare, insertion and lookup are $\mathcal{O}(1)$.

Handling Collisions

The Collision Problem

What happens if two different keys map to the same index? This is a **collision**.

Handling Collisions

The Collision Problem

What happens if two different keys map to the same index? This is a **collision**.

Collision Resolution Strategies

- **Separate Chaining:** Each bucket stores a linked list of entries.
- **Open Addressing (Probing):** If a bucket is full, the algorithm searches (probes) nearby buckets until an empty one is found. (This is what CPython uses).

Handling Collisions

The Collision Problem

What happens if two different keys map to the same index? This is a **collision**.

Collision Resolution Strategies

- **Separate Chaining:** Each bucket stores a linked list of entries.
- **Open Addressing (Probing):** If a bucket is full, the algorithm searches (probes) nearby buckets until an empty one is found. (This is what CPython uses).

Maintaining Performance: Load Factor

The Load Factor is the ratio of entries to buckets. To keep collisions rare and maintain $\mathcal{O}(1)$ performance, the hash table must be resized (rehashed) when the load factor gets too high (e.g., $> 2/3$). This is similar to dynamic array resizing and also uses amortisation.

Dictionaries (`dict`) I

Definition

A dictionary is a mutable mapping of unique keys to values. It implements a hash table.

```python
# Mapping names (keys) to ages (values)
ages = {"Alice": 30, "Bob": 25, "Charlie": 35}

# Accessing a value by key (O(1) average)
bobs_age = ages["Bob"] # 25

# Inserting/Updating (O(1) average, amortized)
ages["David"] = 40

# Membership testing (Checks keys, O(1) average)
is_present = "Alice" in ages
```

Dictionaries (`dict`) II

Definition

A dictionary is a mutable mapping of unique keys to values. It implements a hash table.

Hashability Requirement

Keys must be immutable (hashable) because their hash must remain constant. `int`, `float`, `str`, `tuple` are hashable. `list` and `dict` are mutable and therefore unhashable.

Sets (set) I

Definition

A set is an unordered collection of unique, hashable elements.

```
# Creating a set (duplicates removed automatically)
colors = {"red", "green", "blue", "red"}
# colors is {"red", "green", "blue"}

# Membership testing (O(1) average)
is_present = "red" in colors
```

Sets (set) I

Implementation

A set is implemented as a hash table where only the keys are stored (values are ignored). It inherits the performance characteristics of dictionaries.

```python
# Creating a set (duplicates removed automatically)
colors = {"red", "green", "blue", "red"}
# colors is {"red", "green", "blue"}

# Membership testing (O(1) average)
is_present = "red" in colors

# Adding elements (O(1) average, amortized)
colors.add("yellow")
```

Sets (`set`) II

Definition

A set is an unordered collection of unique, hashable elements.

Implementation

A set is implemented as a hash table where only the keys are stored (values are ignored). It inherits the performance characteristics of dictionaries.

Use Cases

Fast deduplication of lists (`list(set(my_list))`) and efficient membership checking.

Conclusion

Summary of Data Structures

Type	Implementation	Ordered?	Mutable?	Key Performance
`list`	Dynamic Array	Yes	Yes	$\mathcal{O}(1)$ index/append; $\mathcal{O}(N)$ search/insert(0)
`dict`	Hash Table	Yes (since 3.7)	Yes	$\mathcal{O}(1)$ avg lookup/insert/delete
`set`	Hash Table	No	Yes	$\mathcal{O}(1)$ avg membership/add/remove
`tuple`	Fixed Array	Yes	No	$\mathcal{O}(1)$ index; $\mathcal{O}(N)$ search

Key Takeaways

Core Concepts

- **Memory Model:** Python uses references. Understand aliasing and mutability.
- **Implementation Matters:** The underlying data structure determines performance characteristics (Big $\mathcal{O}()$).
- **Dynamic Arrays (Lists):** Optimized for access ($\mathcal{O}(1)$) and operations at the end ($\mathcal{O}(1)$ amortized) via over-allocation. Slow in the middle ($\mathcal{O}(N)$).
- **Amortization:** Geometric growth allows dynamic structures to maintain average $\mathcal{O}(1)$ performance despite occasional $\mathcal{O}(N)$ resizing costs.
- **Hash Tables (Dicts/Sets):** Provide average $\mathcal{O}(1)$ lookups, insertion, and deletion using hashing.

Key Takeaways

The Engineering Trade-off

There is no single best data structure. Choosing the right one depends on the required operations and the expected data patterns.