Python Data Structures Cheat Sheet

Prof. Laio Oriel Seman - laio.seman@ufsc.br

List

Type: Mutable, Ordered

Creation:

```
my_list = []
my_list = [1, 2, 3]
my_list = list()
```

Key Operations:

Operation	Code	Complexity
Access:	`my_list[0]`	O(1)
Append:	`my_list.append(item)`	O(1)
Insert:	`my_list.insert(i, item)`	O(n)
Remove:	`my_list.remove(item)`	O(n)

Use Cases: Dynamic arrays • Stacks • General collections

Tuple

Type: Immutable, Ordered

Creation:

```
my_tuple = ()
my_tuple = (1, 2, 3)
my_tuple = tuple()
```

Key Operations:

Operation	Code	Complexity
Access:	`my_tuple[0]`	O(1)
Count:	`my_tuple.count(item)`	O(n)
Index:	`my_tuple.index(item)`	O(n)
Unpack:	`a, b = my_tuple`	O(1)

Use Cases: Coordinates • Return values • Dict keys

Dictionary

Type: Mutable, Key-Value

Creation:

```
my_dict = {}
my_dict = {'a': 1}
my_dict = dict()
```

Key Operations:

Operation	Code	Complexity
Access:	`my_dict['key']`	O(1)*
Set:	`my_dict['key'] = val`	O(1)*
Delete:	`del my_dict['key']`	O(1)*
Get:	`my_dict.get('key')`	O(1)*

Use Cases: Mappings • Caches • Counting

* Average case. Worst case O(n) due to hash collisions

Set

Type: Mutable, Unique

Creation:

```
my_set = set()
my_set = {1, 2, 3}
my_set = set([1,2])
```

Key Operations:

Operation	Code	Complexity
Add:	`my_set.add(item)`	O(1)*
Remove:	`my_set.remove(item)`	O(1)*
Union:	`set1 | set2`	O(n+m)
Intersect:	`set1 & set2`	O(min(n,m))

Use Cases: Uniqueness • Fast lookup • Set operations

* Average case. Worst case O(n) due to hash collisions

String

Type: Immutable, Sequence

Creation:

```
text = 'hello'
text = "hello"
text = str(123)
```

Key Operations:

Operation	Code	Complexity
Access:	`text[0]`	O(1)
Slice:	`text[1:4]`	O(k)
Find:	`text.find('sub')`	O(n)
Replace:	`text.replace('a','b')`	O(n)

Use Cases: Text processing • Parsing • Formatting

Deque

Type: Double-ended Queue

Creation:

```
from collections import deque
dq = deque()
dq = deque([1,2,3])
```

Key Operations:

Operation	Code	Complexity
AppendLeft:	`dq.appendleft(item)`	O(1)
AppendRight:	`dq.append(item)`	O(1)
PopLeft:	`dq.popleft()`	O(1)
PopRight:	`dq.pop()`	O(1)

Use Cases: Queues • Sliding windows • LRU cache

Time Complexity Comparison

Operation	List	Tuple	Dict	Set	String
Access/Index	O(1)	O(1)	O(1)*	N/A	O(1)
Search	O(n)	O(n)	O(1)*	O(1)*	O(n)
Insert/Add	O(n)	N/A	O(1)*	O(1)*	N/A
Delete	O(n)	N/A	O(1)*	O(1)*	N/A
Append	O(1)	N/A	N/A	N/A	N/A

Common Patterns & Algorithms

Stack Pattern (LIFO)

Using List

```
stack = []
stack.append(item) # push
item = stack.pop() # pop
top = stack[-1] # peek
```

Use Cases:

- Parentheses matching
- DFS
- Undo operations

Queue Pattern (FIFO)

Using Deque

```
from collections import deque
queue = deque()
queue.append(item) # enqueue
item = queue.popleft() # dequeue
```

Use Cases:

- BFS
- Task scheduling
- Buffering

Hash Table Pattern

Using Dict

```
cache = {}
cache[key] = value
if key in cache:
return cache[key]
```

Use Cases:

- Memoization
- Counting
- Fast lookup

Python Iterator Concepts

Container

Definition: Holds multiple items in memory

Features: Supports `in` membership test

Examples: list, set, dict, str, tuple

Code: `3 in [1, 2, 3] → True`

Note: `"foo" in {"foo": 1} → True (key only)`

Iterable

Definition: Object that can return an iterator

Use: Used in for-loops and unpacking

Examples: lists, sets, files, sockets

Code: `iter([1,2,3]) → list_iterator`

Iterator

Definition: Object with `__next__()` and `__iter__()`

Stateful: Remembers position between calls

Examples: `itertools.count(), cycle(), islice()`

Code: `next(iter([1,2,3])) → 1`

Comprehension

Definition: Concise way to create iterables

Syntax: `[x*x for x in range(3)]`

Behavior: Eager, evaluates all at once

Code: `[x*x for x in range(3)] → [0, 1, 4]`

Generator Function

Definition: Special iterator using `yield`

Advantage: Lazy, compact, elegant syntax

Example: `def fib(): yield 1; yield 2`

Usage: `g = fib(); next(g) → 1`

Generator Expression

Definition: Compact syntax for lazy generation

Syntax: `(x*x for x in range(3))`

Behavior: Like generator, consumes values

Code: `next(g) → 0; list(g) → [1, 4]`

Data Structures

Binary Tree

Tree structure with at most 2 children per node

Key Operations:

Operation	Code	Complexity
Insert:	`insert(node, value)`	O(log n)
Search:	`search(node, value)`	O(log n)
Delete:	`delete(node, value)`	O(log n)
Traversal:	`inorder(node)preorder(node)postorder(node)`	O(n)

Linked List

Sequence of nodes where each node points to the next

Key Operations:

Operation	Code	Complexity
Insert:	`insert(head, value)`	O(1)
Search:	`search(head, value)`	O(n)
Delete:	`delete(head, value)`	O(n)
Traversal:	`traverse(head)`	O(n)

Doubly Linked List

Linked list where each node points to both next and previous nodes

Key Operations:

Operation	Code	Complexity
Insert:	`insert(head, value)`	O(1)
Search:	`search(head, value)`	O(n)
Delete:	`delete(head, value)`	O(n)
Traversal:	`traverse(head)`	O(n)

Heap

Complete binary tree where parent is greater (max-heap) or smaller (min-heap) than children

Key Operations:

Operation	Code	Complexity
Insert:	`insert(heap, value)`	O(log n)
Extract Max/Min:	`extract_max(heap)`	O(log n)
Peek:	`peek(heap)`	O(1)
Heapify:	`heapify(arr)`	O(n)

Graph

Collection of nodes (vertices) connected by edges

Key Operations:

Operation	Code	Complexity
Add Vertex:	`add_vertex(graph, vertex)`	O(1)
Add Edge:	`add_edge(graph, v1, v2)`	O(1)
Remove Vertex:	`remove_vertex(graph, vertex)`	O(V + E)
Remove Edge:	`remove_edge(graph, v1, v2)`	O(E)
Traversal:	`dfs(graph, start)bfs(graph, start)`	O(V + E)

Priority Queue

Abstract data type where each element has a priority

Key Operations:

Operation	Code	Complexity
Insert:	`insert(pq, item, priority)`	O(log n)
Extract Max/Min:	`extract_max(pq)`	O(log n)
Peek:	`peek(pq)`	O(1)
Is Empty:	`is_empty(pq)`	O(1)

Sorting Algorithms Comparison

Bubble Sort

Repeatedly swaps adjacent elements if they're in wrong order

Time Complexity:
Best: O(n)
Average: $O(n^2)$
Worst: $O(n^2)$
Space: O(1)

Properties:
Stable: ✓
Adaptive: ✓

Implementation:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True
        if not swapped:
            break
```

Insertion Sort

Builds sorted array one element at a time

Time Complexity:
Best: O(n)
Average: $O(n^2)$
Worst: $O(n^2)$
Space: O(1)

Properties:
Stable: ✓
Adaptive: ✓

Implementation:

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
```

Selection Sort

Selects minimum element and swaps with first unsorted

Time Complexity:
Best: $O(n^2)$
Average: $O(n^2)$
Worst: $O(n^2)$
Space: O(1)

Properties:
Stable: ✗
Adaptive: ✗

Implementation:

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
```

Quick Sort

Divide-and-conquer with pivot partitioning

Time Complexity:
Best: O(n log n)
Average: O(n log n)
Worst: $O(n^2)$
Space: O(log n)

Properties:
Stable: ✗
Adaptive: ✗

Implementation:

```
def quick_sort(arr, low=0, high=None):
    if high is None: high = len(arr) - 1
    if low < high:
        pi = partition(arr, low, high)
        quick_sort(arr, low, pi - 1)
        quick_sort(arr, pi + 1, high)

def partition(arr, low, high):
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        if arr[j] <= pivot:
```

Sorting Algorithm Complexity Summary

Algorithm	Best	Average	Worst	Space	Stable
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	✓
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	✓
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	✗
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$	O(log n)	✗
Merge Sort*	O(n log n)	O(n log n)	O(n log n)	O(n)	✓
Heap Sort*	O(n log n)	O(n log n)	O(n log n)	O(1)	✗

Algorithm	Best	Average	Worst	Space	Stable
Tim Sort	O(n)	O(n log n)	O(n log n)	O(n)	✓
Shell Sort	O(n log n)	O(n (log n)^2)	$O(n^2)$	O(1)	✗
Radix Sort*	O(nk)	O(nk)	O(nk)	O(n + k)	✓
Bucket Sort*	O(n+k)	O(n+k)	$O(n^2)$	O(n)	✓
Counting Sort*	O(n+k)	O(n+k)	O(n+k)	O(k)	✓
Bogo Sort	O(n)	O(n·n!)	O(∞)	O(1)	✗

* Algorithms with specialized assumptions or auxiliary structures

Graph Algorithms

Depth-First Search

Explores as far as possible before backtracking

Time: O(V + E)
Space: O(V)

Implementation:

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
    return visited
```

Breadth-First Search

Explores level by level using queue

Time: O(V + E)
Space: O(V)

Implementation:

```
from collections import deque

def bfs(graph, start):
    visited = set([start])
    queue = deque([start])
    while queue:
        vertex = queue.popleft()
        for neighbor in graph[vertex]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
    return visited
```

Dijkstra's Algorithm

Finds shortest path in weighted graphs

Time: O((V + E) log V)
Space: O(V)

Implementation:

```
import heapq

def dijkstra(graph, start):
    queue = [(0, start)]  # (distance, vertex)
    distances = {vertex: float('infinity') for vertex in graph}
    distances[start] = 0

    while queue:
        current_distance, current_vertex = heapq.heappop(queue)

        if current_distance > distances[current_vertex]:
            continue

        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(queue, (distance, neighbor))

    return distances
```

Practical Examples

Counting Elements

```
# Using dict for counting
words = ['apple', 'banana', 'apple']
count = {}
for word in words:
    count[word] = count.get(word, 0) + 1
# Result: {'apple': 2, 'banana': 1}
```

Remove Duplicates

```
# Using set to remove duplicates
numbers = [1, 2, 2, 3, 3, 4]
unique = list(set(numbers))
# Result: [1, 2, 3, 4]

# Preserve order with dict
unique_ordered = list(dict.fromkeys(numbers))
```

Stack for Parentheses

```
def is_valid_parentheses(s):
    stack = []
    pairs = {'(': ')', '[': ']', '{': '}'}
    for char in s:
        if char in pairs:
            stack.append(char)
        elif char in pairs.values():
            if not stack or pairs[stack.pop()] != char:
                return False
    return len(stack) == 0
```

BFS with Queue

```
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])
    while queue:
        node = queue.popleft()
        if node not in visited:
            visited.add(node)
            queue.extend(graph[node])
```

LRU Cache Implementation

```
from collections import OrderedDict

class LRUCache:
    def __init__(self, capacity):
        self.cache = OrderedDict()
        self.capacity = capacity
    def get(self, key):
        if key in self.cache:
            self.cache.move_to_end(key)
            return self.cache[key]
```

Two Pointers Technique

```
def two_sum_sorted(nums, target):
    left, right = 0, len(nums) - 1
    while left < right:
        current_sum = nums[left] + nums[right]
        if current_sum == target:
            return [left, right]
        elif current_sum < target:
            left += 1
        else:
            right -= 1
```

Sliding Window Max

```
from collections import deque

def sliding_window_max(nums, k):
    dq = deque()
    result = []
    for i, num in enumerate(nums):
        while dq and dq[0] <= i - k:
            dq.popleft()
        while dq and nums[dq[-1]] <= num:
            dq.pop()
```

Trie Data Structure

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end = False

class Trie:
    def __init__(self):
        self.root = TrieNode()
    def insert(self, word):
        node = self.root
```

When to Use What?

Question	Choice	Next
Need to store key-value pairs?	✓ Use Dictionary	→ Continue...
Need unique elements only?	✓ Use Set	→ Continue...
Data will never change?	✓ Use Tuple	→ Continue...
Need frequent insertions at both ends?	✓ Use Deque	→ Continue...
Working with text/characters?	✓ Use String	→ Use List

Memory & Performance Tips

- Lists: Use for dynamic arrays, avoid frequent insertions at beginning
- Tuples: Use for immutable data, coordinates, function returns
- Dicts: Use for O(1) lookups, counting, caching
- Sets: Use for uniqueness checks, fast membership testing
- Strings: Immutable - concatenation creates new objects
- Deque: Use for queues and double-ended operations
- List comprehensions are faster than loops for creation
- 'in' operator: O(1) for dicts/sets, O(n) for lists/tuples
- Pre-allocate list size if known: [None] * n

Choose the right data structure for the job!

Performance matters: Know your complexities • Practice with real problems