Python Notes for Data Structures & Algorithms (DSA)

Welcome back to DSA! Since you have a background in C++, C, Java, and JavaScript, you'll find Python's simplicity and rich standard library a huge advantage for DSA. These notes will highlight key Python features and best practices for problem-solving.

1. Core Python Features for DSA

Python's built-in data types and structures are incredibly powerful and often abstract away complexities you'd handle manually in C++ or Java.

1.1. Basic Data Types

- **Integers (int):** Arbitrary precision (no overflow issues like in C++/Java int).
  ```
  a = 10
  b = 10**100 # Very large integer, no problem
  ```
- **Floats (float):** Standard double-precision floating-point numbers.
- **Booleans (bool):** True or False.
- **Strings (str):** Immutable sequences of characters.
  - **Concatenation:** s1 + s2
  - **Slicing:** s[start:end:step] (e.g., s[::-1] for reverse)
  - **Common methods:** len(s), s.lower(), s.upper(), s.strip(), s.split(), s.join(list_of_strings)

1.2. Essential Built-in Data Structures

1.2.1. Lists (list) - Your Dynamic Array / Stack / Queue

Lists are Python's most versatile sequence type. They are dynamic arrays, meaning they can grow and shrink in size.

- **Creation:**
  ```
  my_list = []
  my_list = [1, 2, "hello", True]
  ```
- **Accessing Elements:**
  ```
  print(my_list[0])    # 1
  print(my_list[-1])   # True (last element)
  ```
- **Common Operations:**
  - append(element): Add to the end (Amortized O(1)).
  - pop(): Remove and return last element (O(1)).

- pop(index): Remove and return element at index (O(N)).
- insert(index, element): Insert at index (O(N)).
- remove(value): Remove first occurrence of value (O(N)).
- len(my_list): Get length (O(1)).
- sort(): Sorts in-place (O(N log N)). sorted(list) returns a new sorted list.
- reverse(): Reverses in-place (O(N)). list[::-1] creates a new reversed list.
- in operator: element in my_list (O(N) for lists).

- **Slicing:** Powerful for sub-lists.
```
sub_list = my_list[1:3] # [2, "hello"]
copy_list = my_list[:]  # Creates a shallow copy
```

- **Using Lists as Stacks:**
```
stack = []
stack.append(1) # Push
stack.append(2)
top_element = stack.pop() # Pop
```

- **Using Lists as Basic Queues (Inefficient for large queues):**
```
queue = []
queue.append(1) # Enqueue
queue.append(2)
front_element = queue.pop(0) # Dequeue (O(N) - avoid for large queues)
```

1.2.2. Tuples (tuple) - Immutable Sequences

Tuples are similar to lists but are immutable. Once created, their elements cannot be changed.

- **Creation:**
```
my_tuple = (1, 2, "a")
single_element_tuple = (5,) # Comma is essential for single-element tuples
```

- **Use Cases:** Often used for fixed collections of items, function return values (e.g., returning (value, index)), and as dictionary keys (because they are hashable due to immutability).

1.2.3. Sets (set) - Unordered Collections of Unique Elements

Sets are useful for quickly checking membership, removing duplicates, and performing set operations.

- **Creation:**

```
my_set = {1, 2, 3, 2} # my_set will be {1, 2, 3}
empty_set = set()
```

- **Common Operations (mostly O(1) on average):**
  - add(element)
  - remove(element) (raises KeyError if not found)
  - discard(element) (no error if not found)
  - element in my_set (fast membership test)
  - union(), intersection(), difference()

1.2.4. Dictionaries (dict) - Your Hash Map / Hash Table

Dictionaries store key-value pairs and provide very fast lookups based on keys.

- **Creation:**
  ```
  my_dict = {"name": "Alice", "age": 30}
  empty_dict = {}
  ```

- **Accessing/Modifying:**
  ```
  print(my_dict["name"]) # "Alice"
  my_dict["city"] = "New York"
  ```

- **Common Operations (mostly O(1) on average):**
  - my_dict[key]: Access value (raises KeyError if key not found).
  - my_dict.get(key, default_value): Safely get value, returns default_value if key not found (default is None).
  - key in my_dict: Check if key exists (fast membership test).
  - del my_dict[key]: Delete a key-value pair.
  - my_dict.keys(): Returns a view of keys.
  - my_dict.values(): Returns a view of values.
  - my_dict.items(): Returns a view of key-value pairs.
- **Iteration:**
  ```
  for key, value in my_dict.items():
     print(f"{key}: {value}")
  ```

1.3. Control Flow

- **if/elif/else:** Standard conditional statements.
  ```
  if x > 0:
     print("Positive")
  elif x < 0:
  ```

```python
        print("Negative")
    else:
        print("Zero")
```

- **for loops:** Iterate over sequences (lists, strings, tuples, dictionaries).

```python
for item in my_list:
    print(item)
for i in range(5): # 0, 1, 2, 3, 4
    print(i)
for i in range(len(my_list)): # Iterate with index
    print(my_list[i])
```

- **while loops:** Loop as long as a condition is true.

```python
count = 0
while count < 5:
    print(count)
    count += 1
```

1.4. Functions

Define reusable blocks of code.

```python
def greet(name):
    return f"Hello, {name}!"

message = greet("World")
print(message)
```

1.5. Classes (Basic Object-Oriented Programming)

Essential for implementing custom data structures like Linked Lists, Trees, Graphs, etc.

```python
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None # For linked lists
        self.left = None # For trees
        self.right = None # For trees

    def __str__(self): # Optional: for easy printing
```

```python
        return f"Node({self.value})"

# Example usage:
node1 = Node(10)
node2 = Node(20)
node1.next = node2
print(node1) # Node(10)
print(node1.next) # Node(20)
```

1.6. Recursion

A function calling itself. Crucial for many tree and graph algorithms.

```python
def factorial(n):
    if n == 0: # Base case
        return 1
    else: # Recursive step
        return n * factorial(n - 1)

print(factorial(5)) # 120
```

1.7. List Comprehensions

A concise way to create lists.

```python
squares = [x**2 for x in range(10) if x % 2 == 0] # [0, 4, 16, 36, 64]
```

2. DSA Specifics in Python

2.1. Time and Space Complexity Analysis

Understanding Big O notation is critical. Python's built-in operations have specific complexities:

- **len(list) / len(dict) / len(set):** O(1)
- **List append() / pop() (from end):** Amortized O(1)
- **List insert() / pop(0) / remove():** O(N) (because elements need to be shifted)
- **element in list:** O(N) (linear search)
- **key in dict / element in set:** Average O(1), Worst O(N) (due to hash collisions, rare in practice)

- **Dictionary/Set insertion/deletion:** Average O(1), Worst O(N)

2.2. Common DSA Patterns & Pythonic Ways

- **Efficient Queues:** For true O(1) enqueue and dequeue from both ends, use collections.deque.
```
from collections import deque

q = deque()
q.append(1) # Add to right (enqueue)
q.append(2)
print(q.popleft()) # Remove from left (dequeue) - O(1)
```

- **Heaps (Priority Queues):** Use the heapq module. Python's heapq implements a min-heap.
```
import heapq

min_heap = []
heapq.heappush(min_heap, 3)
heapq.heappush(min_heap, 1)
heapq.heappush(min_heap, 5)
print(heapq.heappop(min_heap)) # 1 (smallest element)
```

- **Graphs:** Typically represented using an **adjacency list** (dictionary where keys are nodes and values are lists/sets of neighbors).
```
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D'],
    'C': ['A', 'E'],
    'D': ['B'],
    'E': ['C']
}
# Or using sets for faster neighbor lookup if order doesn't matter
graph_set = {
    'A': {'B', 'C'},
    'B': {'A', 'D'},
    'C': {'A', 'E'},
    'D': {'B'},
    'E': {'C'}
}
```

- **Trees:** Implemented using custom Node classes, as shown in the Classes section.

2.3. Input/Output

- **Reading input:**
```
s = input() # Reads a line as a string
n = int(input()) # Reads a line and converts to integer
nums = list(map(int, input().split())) # Reads space-separated integers into a list
```

- **Printing output:**
```
print("Hello")
print(f"The answer is {result}") # f-strings are great for formatting
```

3. Tips for DSA Practice in Python

1. **Leverage Built-ins:** Don't re-invent the wheel unless the problem specifically asks for it. Use list, dict, set, collections.deque, heapq where appropriate.
2. **Understand Underlying Implementations:** While Python abstracts details, knowing that list.pop(0) is O(N) is crucial for choosing deque when a true queue is needed.
3. **Practice Regularly:** Consistency is key. Start with easy problems on platforms like LeetCode and gradually move to medium and hard ones.
4. **Focus on Logic First:** Get a working (even if inefficient) solution first. Then, think about optimization (time and space complexity).
5. **Use collections Module:** Besides deque, explore collections.Counter (for frequency counting) and collections.defaultdict (for dictionaries with default values).
6. **Read Solutions:** After attempting a problem, always review optimal solutions, even if you solved it. Learn new approaches and Pythonic tricks.
7. **Write Clean Code:** Python's readability helps here. Use meaningful variable names.

By focusing on these Pythonic approaches, you'll find your DSA practice much more efficient and enjoyable compared to the lower-level complexities of C++. Good luck!