8 Data Structures Every Python Programmer Should Know

The Educative Team

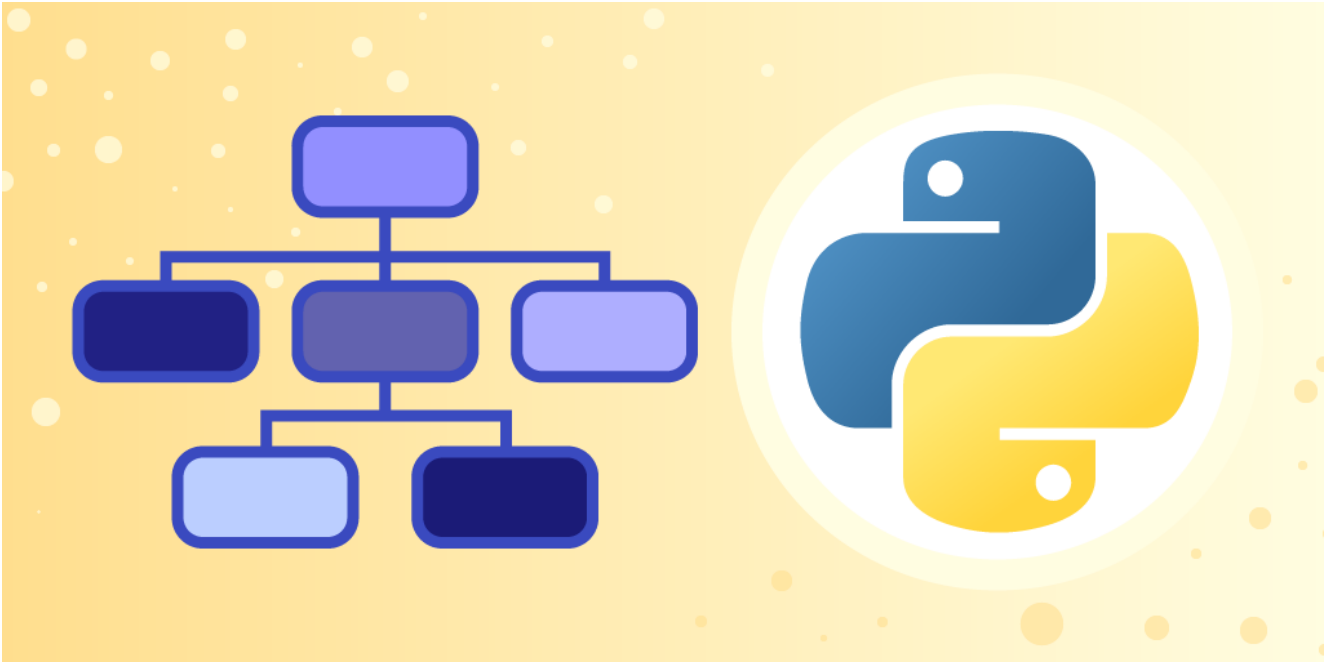

Image Source: Author

When solving real-world coding problems, employers and recruiters are looking for both runtime and resource efficiency.

Knowing which data structure best fits the current solution will increase program performance and reduce the time required to make it. For this reason, most top companies require a strong understanding of data structures and heavily test them in their coding interview.

Here's what we'll cover today:

- What are data structures?
- Arrays in Python
- Queues in Python
- Stacks in Python
- Linked lists in Python
- Circular linked lists in Python
- Trees in Python
- Graphs in Python
- Hash tables in Python
- What to learn next

What are data structures?

Data structures are code structures for storing and organizing data that make it easier to modify, navigate, and access information. Data structures determine how data is collected, the functionality we can implement, and the relationships between data.

Data structures are used in almost all areas of computer science and programming, from operating systems, to front-end development, to machine learning.

Data structures help to:

- Manage and utilize large datasets
- Quickly search for particular data from a database
- Build clear hierarchical or relational connections between data points
- Simplify and speed up data processing

Data structures are **vital building blocks** for efficient, real-world problem-solving. Data structures are proven and optimized tools that give you an easy frame to organize your programs. After all, there's no need for you to remake the wheel (or structure) every time you need it.

Each data structure has a task or situation it is **most suited to solve**. Python has 4 built-in data structures, lists, dictionaries, tuples, and sets. These built-in data structures come with default methods and behind-the-scenes optimizations that make them easy to use.

Most data structures in Python have modified forms of these or use the built-in structures as their backbone.

- **List**: Array-like structures that let you save a set of mutable objects of the same type to a variable.
- **Tuple**: Tuples are immutable lists, meaning the elements cannot be changed. It's declared with parenthesis instead of square brackets.
- **Set**: Sets are unordered collections, meaning that elements are unindexed and have no set sequence. They're declared with curly braces.
- **Dictionary (dict)**: Similar to hashmap or hash tables in other languages, a dictionary is a collection of key/value pairs. You initialize an empty dictionary with empty curly braces and fill it with colon-separated keys and values. All keys are unique, immutable objects.

Now, let's see how we can use these structures to create all the advanced structures interviewers are looking for.

Arrays (Lists) in Python

Python does not have a built-in array type, but you can use lists for all of the same tasks. An array is a collection of values of the **same type saved under the same name**.

Each value in the array is called an "element" and indexing that represents its position. You can access specific elements by calling the array name with the desired element's index. You can also get the length of an array using the `len()` method.

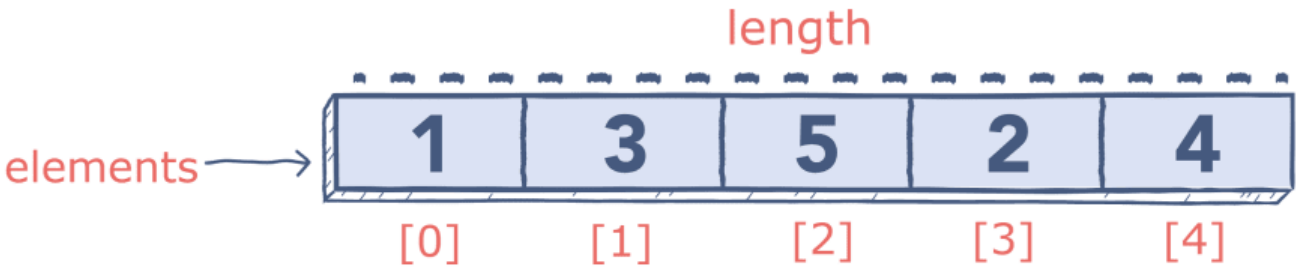

Image Source: Author

Unlike programming languages like Java that have static arrays after declaration, Python's arrays **automatically scale up or down** when elements are added/subtracted.

For example, we could use the `append()` method to add an additional element on the end of an existing array instead of declaring a new array.

This makes Python arrays particularly easy to use and

adaptable on the fly.

```
cars = ["Toyota", "Tesla", "Hyundai"]
print(len(cars))
cars.append("Honda")
cars.pop(1)
for x in cars:
  print(x)
```

Advantages:

- Simple to create and use data sequences
- Automatically scale to meet changing size requirements
- Used to create more complex data structures

Disadvantages:

- Not optimized for scientific data (unlike NumPy's array)
- Can only manipulate the rightmost end of the list

Applications:

- Shared storage of related values or objects, i.e. `myDogs`
- Data collections you'll loop through
- Collections of data structures, such as a list of tuples

Common arrays interview questions

in Python

- Remove even integers from a list
- Merge two sorted lists
- Find the minimum value in a list
- Maximum sum sublist
- Print products of all elements

Queues in Python

Queues are a linear data structure that store data in a "first-in, first-out" (FIFO) order. Unlike arrays, you cannot access elements by index and instead can **only pull the next oldest element**. This makes it great for order-sensitive tasks like online order processing or voicemail storage.

You can think of a queue as a line at the grocery store; the cashier does not choose who to check out next but rather processes the person who has stood in line the longest.

Image Source: Author

We could use a Python list with `append()` and `pop()` methods to implement a queue. However, this is inefficient because lists must shift all elements by one index whenever you add a new element to the beginning.

Instead, it's best practice to use the `deque` class from Python's `collections` module. Deques are optimized for the append and pop operations. The deque implementation also allows you to create double-ended queues, which can access both sides of the queue through the `popleft()` and `popright()` methods.

```
from collections import deque# Initializing a que
q = deque()# Adding elements to a queue
q.append('a')
q.append('b')
q.append('c')print("Initial queue")
print(q)# Removing elements from a queue
print("\nElements dequeued from the queue")
print(q.popleft())
print(q.popleft())
print(q.popleft())print("\nQueue after removing e
print(q)# Uncommenting q.popleft()
# will raise an IndexError
# as queue is now empty
```

Advantages:

- Automatically orders data chronologically
- Scales to meet size requirements
- Time-efficient with `deque` class

Disadvantages:

- Can only access data on the ends

Applications:

- Operations on a shared resource like a printer or CPU core
- Serve as temporary storage for batch systems
- Provides an easy default order for tasks of equal importance

Common queue interview questions in Python

- Reverse first k elements of a queue
- Implement a queue using a linked list
- Implement a stack using a queue

Stacks in Python

Stacks are a sequential data structure that acts as the Last-in, First-out (LIFO) version of queues. The last element inserted in a stack is considered at the **top of the stack** and is the only accessible element. To access a middle element, you must first remove enough elements to make the desired element at the top of the stack.

Many developers imagine stacks as a stack of dinner plates; you can add or remove plates to the top of the

stack but must move the whole stack to place one at the bottom.

Image Source: Author

Adding elements is known as a **push,** and removing elements is known as a **pop**. You can implement stacks in Python using the built-in list structure. With list implementation, push operations use the `append()` method, and pop operations use `pop()`.

```
stack = []# append() function to push
# element in the stack
```

```
stack.append('a')
stack.append('b')
stack.append('c')print('Initial stack')
print(stack)# pop() function to pop
# element from stack in
# LIFO order
print('\nElements popped from stack:')
print(stack.pop())
print(stack.pop())
print(stack.pop())print('\nStack after elements a
print(stack)# uncommenting print(stack.pop())
# will cause an IndexError
# as the stack is now empty
```

Advantages:

- Offers LIFO data management that’s impossible with *Applications:*s or arrays
- Automatic scaling and object cleanup
- Simple and reliable data storage system

Disadvantages:

- Stack memory is limited
- Too many objects on the stack leads to a stack overflow error

Applications:

- Used for making highly reactive systems
- Memory management systems use stacks to handle the most recent requests first

- Helpful for questions like parenthesis matching

Common stacks interview questions in Python

- Implement a queue using stacks
- Evaluate a Postfix expression with a stack
- Next greatest element using a stack
- Create a `min()` function using a stack

Linked lists in Python

Linked lists are a sequential collection of data that uses **relational pointers on each data node** to link to the next node in the list.

Unlike arrays, linked lists do not have objective positions in the list. Instead, they have relational positions based on their surrounding nodes.

The first node in a linked list is called the **head node,** and the final is called the **tail node**, which has a `null` pointer.

Image Source: Author

Linked lists can be singly or doubly linked depending if each node has just a single pointer to the next node or if it also has a second pointer to the previous node.

You can think of linked lists like a chain; individual links only have a connection to their immediate neighbors but all the links together form a larger structure.

Python does not have a built-in implementation of linked lists and therefore requires that you implement a `Node` class to hold a data value and one or more pointers.

```
class Node:
    def __init__(self, dataval=None):
        self.dataval = dataval
        self.nextval = Noneclass SLinkedList:
```

```
    def __init__(self):
        self.headval = Nonelist1 = SLinkedList()
list1.headval = Node("Mon")
e2 = Node("Tue")
e3 = Node("Wed")
# Link first Node to second node
list1.headval.nextval = e2# Link second Node to t
e2.nextval = e3
```

Linked lists are primarily used to create advanced data structures like graphs and trees or for tasks that require frequent addition/deletion of elements across the structure.

Advantages:

- Efficient insertion and deletion of new elements
- Simpler to reorganize than arrays
- Useful as a starting point for advanced data structures like graphs or trees

Disadvantages:

- Storage of pointers with each data point increases memory usage
- Must always traverse the linked list from Head node to find a specific element

Applications:

- Building block for advanced data structures
- Solutions that call for frequent addition and removal

of data

Common linked list interview questions in Python

- Print the middle element of a given linked list
- Remove duplicate elements from a sorted linked list
- Check if a singly linked list is a palindrome
- Merge K sorted linked lists
- Find the intersection point of two linked lists

Circular linked lists in Python

The primary downside of the standard linked list is that you always have to start at the Head node. The circular linked list fixes this problem by replacing the Tail node's `null` pointer with a pointer back to the Head node. When traversing, the program will follow pointers until it reaches the node it started on.

Image Source: Author

The advantage of this setup is that you can start at any node and traverse the whole list. It also allows you to use linked lists as a loopable structure by setting a desired number of cycles through the structure. Circular linked lists are great for processes that loop for a long time like CPU allocation in operating systems.

Advantages:

- Can traverse the whole list starting from any node
- Makes linked lists more suited to looping structures

Disadvantages:

- More difficult to find the Head and Tail nodes of the list without a `null` marker

Applications:

- Regularly looping solutions like CPU scheduling

Common circular linked list interview questions in Python

- Detect loop in a linked lists
- Reverse a circular linked list
- Reverse circular linked list in groups of a given size

Trees in Python

Trees are another relation-based data structure, which specialize in representing hierarchical structures. Like a linked list, they're populated with `Node` objects that contain a data value and one or more pointers to define its relation to immediate nodes.

Each tree has a **root** node that all other nodes branch off from. The root contains pointers to all elements directly below it, which are known as its **child nodes**. These child nodes can then have child nodes of their own. Binary trees cannot have nodes with more than two child nodes.

Any nodes on the same level are called **sibling nodes**. Nodes with no connected child nodes are known as **leaf nodes**.

The most common application of the binary tree is a **binary search tree**. Binary search trees excel at searching large collections of data, as the time complexity depends on the depth of the tree rather than the number of nodes.

Binary search trees have four strict rules:

- The left subtree contains only nodes with elements lesser than the root.
- The right subtree contains only nodes with elements greater than the root.
- Left and right subtrees must also be a binary search tree. They must follow the above rules with the “root”

of their tree.

- There can be no duplicate nodes, i.e. no two nodes can have the same value.

```
class Node:    def __init__(self, data):          s
        self.right = None
        self.data = data    def insert(self, data
# Compare the new value with the parent node
        if self.data:
            if data < self.data:
                if self.left is None:
                    self.left = Node(data)
                else:
                    self.left.insert(data)
            elif data > self.data:
                if self.right is None:
                    self.right = Node(data)
                else:
                    self.right.insert(data)
        else:
            self.data = data# Print the tree
    def PrintTree(self):
        if self.left:
            self.left.PrintTree()
        print( self.data),
        if self.right:
            self.right.PrintTree()# Use the inser
root = Node(12)
root.insert(6)
root.insert(14)
root.insert(3)root.PrintTree()
```

Advantages:

- Good for representing hierarchical relationships
- Dynamic size, great at scale
- Quick insert and delete operations
- In a binary search tree, inserted nodes are sequenced immediately.
- Binary search trees are efficient at searches; length is only *O(height)*.

Disadvantages:

Time expensive, *O(logn)*, to modify or "balance" trees or retrieve elements from a known location

- Child nodes hold no information on their parent node and can be hard to traverse backward
- Only works for lists that are sorted. Unsorted data degrades into linear search.

Applications:

- Great for storing hierarchical data such as a file location
- Used to implement top searching and sorting algorithms like binary search trees and binary heaps

Common tree interview questions in Python

- Check if two binary trees are identical
- Implement level order traversal of a binary tree
- Print the perimeter of a binary search tree
- Sum all nodes along a path
- Connect all siblings of a binary tree

Graphs in Python

Graphs are a data structure used to represent a visual of relationships between data **vertices** (the Nodes of a graph). The links that connect vertices together are called **edges**.

Edges define which vertices are connected but do not indicate a direction flow between them. Each vertex has connections to other vertices which are saved on the vertex as a comma-separated list.

Image Source: Author

There are also special graphs called **directed graphs** that define a direction of the relationship, similar to a linked list. Directed graphs are helpful when modeling one-way relationships or a flowchart-like structure.

Image Source: Author

They’re primarily used to convey visual web-structure networks in code form. These structures can model many different types of relationships like hierarchies, branching structures, or simply be an unordered relational web. The versatility and intuitiveness of graphs makes them a favorite for data science.

When written in plain text, graphs have a list of vertices and edges:

```
V = {a, b, c, d, e}
E = {ab, ac, bd, cd, de}
```

In Python, graphs are best implemented using a dictionary with the name of each vertex as a key and the edges list as the values.

```
# Create the dictionary with graph elements
graph = { "a" : ["b","c"],
                  "b" : ["a", "d"],
                  "c" : ["a", "d"],
                   "d" : ["e"],
                   "e" : ["d"]
          }# Print the graph
print(graph)
```

Advantages:

- Quickly convey visual information through code
- Usable for modeling a wide range of real-world problems
- Simple to learn the syntax

Disadvantages:

- Vertex links are difficult to understand in large graphs
- Time expensive to parse data from a graph

Applications:

- Excellent for modeling networks or web-like

structures
- Used to model social network sites like Facebook

Common graph interview questions in Python

- Detect cycle in a directed graph
- Find a “Mother Vertex” in a directed graph
- Count the number of edges in an undirected graph
- Check if a path exists between two vertices
- Find the shortest path between two vertices

Hash tables in Python

Hash tables are a complex data structure capable of storing large amounts of information and retrieving specific elements efficiently.

This data structure uses key/value pairs, where the key is the name of the desired element and the value is the data stored under that name.

Image Source: Author

Each input key goes through a **hash function** that converts it from its starting form into an integer value, called a **hash**. Hash functions must always produce the same hash from the same input, must compute quickly, and produce fixed-length values. Python includes a built-in `hash()` function that speeds up implementation.

The table then uses the hash to find the general location of the desired value, called a **storage bucket**. The program then only has to search this subgroup for the desired value rather than the entire data pool.

Beyond this general framework, hash tables can be very different depending on the application. Some may allow keys from different data types, while some may have differently setup buckets or different hash functions.

Here is an example of a hash table in Python code:

```
import pprint
```

```
class Hashtable:
    def __init__(self, elements):
        self.bucket_size = len(elements)
        self.buckets = [[] for i in range(self.bu
        self._assign_buckets(elements)
    def _assign_buckets(self, elements):
        for key, value in elements: #calculates t
            hashed_value = hash(key)
            index = hashed_value % self.bucket_si
            self.buckets[index].append((key, valu
    def get_value(self, input_key):
        hashed_value = hash(input_key)
        index = hashed_value % self.bucket_size
        bucket = self.buckets[index]
        for key, value in bucket:
            if key == input_key:
                return(value)
        return None
    def __str__(self):
        return pprint.pformat(self.buckets) # pfo
if __name__ == "__main__":
     capitals = [
        ('France', 'Paris'),
        ('United States', 'Washington D.C.'),
        ('Italy', 'Rome'),
        ('Canada', 'Ottawa')
    ]
hashtable = Hashtable(capitals)
print(hashtable)
print(f"The capital of Italy is {hashtable.get_va
```

Advantages:

- Can covert keys in any form to integer indices
- Extremely effective for large data sets
- Very effective search function
- Constant number of steps for each search and constant efficiency for adding or deleting elements
- Optimized in Python 3

Disadvantages:

- Hashes must be unique, two keys converting to the same hash causes a collision error
- Collision errors require a full overhaul of the hash function
- Difficult to build for beginners

Applications:

- Used for large, frequently-searched databases
- Retrieval systems that use input keys

Common hash table interview questions in Python

- Build a hash table from scratch (without built-in functions)
- Word formation using a hash table
- Find two numbers that add up to “k”
- Implement open addressing for collision handling
- Detect if a list is cyclical using a hash table

What to learn next

There are dozens of interview questions and formats for each of these 8 data structures. The best way to prepare yourself for the interview process is to keep trying hands-on practice problems.

Here's some beginner questions to get you started:

- Reverse a string in Python
- Check if a Python string contains another substring
- Implement breadth-first search of a binary tree
- Merge two sorted lists
- Find the first non-repeating element in a list of integers

Happy learning!

Find more content at ***plainenglish.io***