



Python Data Structure EBook

Copyright: CodewithCurious.com

Copyrighted By @Curious_.programmer

Python Data Structures

Python provides a variety of built-in data structures that can be used to store and manipulate data. This index page lists the most commonly used data structures in Python and provides links to detailed explanations and examples.

● Lists:

Lists are one of the most versatile data structures in Python, and are used to store collections of items. Lists can contain elements of different data types and can be modified after they are created.

- Introduction to Lists :
- List Methods
- List Comprehensions

● Tuples :

Tuples are similar to lists in that they can store collections of items, but they are immutable and cannot be modified once they are created.

- Introduction to Tuples :
- Tuple Methods

● Sets :

Sets are used to store collections of unique items. Sets can be modified after they are created and support operations such as union, intersection, and difference.

- Introduction to Sets
- Set Methods

● Dictionaries

PDF FILE UPLOADED ON TELEGRAM {Link in Bio}

Dictionaries are used to store collections of key-value pairs. Dictionaries are mutable and can be modified after they are created.

- Introduction to Dictionaries
- Dictionary Methods

● **Arrays :**

Arrays are used to store collections of homogeneous items, and can be more efficient than lists for certain types of operations.

- Introduction to Arrays
- Array Methods

● **Queues :**

Queues are used to store collections of items in a first-in, first-out (FIFO) order.

- Introduction to Queues
- Queue Methods

● **Stacks :**

Stacks are used to store collections of items in a last-in, first-out (LIFO) order.

- Introduction to Stacks
- Stack Methods

● **Linked Lists :**

Linked lists are a data structure in which each element (or node) contains a reference to the next element in the list.

- Introduction to Linked Lists
- Linked List Methods

● **Trees :**

Trees are used to store hierarchical data, and can be used to represent things like file systems and organizational charts.

- Introduction to Trees
- Tree Traversal Methods

● **Graphs :**

Graphs are used to represent networks of nodes and edges.

- Introduction to Graphs
- Graph Traversal Methods

This is just a sampling of the many data structures available in Python, but it should give you a good starting point for learning more about how to use them.

Python Data Structures

Python is a powerful and popular programming language that is widely used in a variety of applications. One of the key features of Python is its support for a wide range of data structures that make it easy to organize and manipulate data in different ways. In this article, we will discuss some of the most commonly used data structures in Python and how to use them.

Python provides a variety of built-in data structures, such as lists, tuples, sets, and dictionaries, that make it easy to organize and manipulate data.

1. Lists are ordered collections of elements that can be modified after they are created.
2. Tuples are similar to lists, but they are immutable, which means that they cannot be modified after they are created.
3. Sets are used to store collections of unique items, and they support operations such as union, intersection, and difference.
4. Dictionaries are used to store key-value pairs and allow for efficient lookup of values based on their associated keys.
5. Arrays are used to store collections of homogeneous items, and they can be more efficient than lists for certain types of operations.
6. Queues and stacks are used to store collections of items in a first-in, first-out (FIFO) or last-in, first-out (LIFO) order, respectively.
7. Linked lists are a data structure in which each element (or node) contains a reference to the next element in the list.
8. Trees are used to store hierarchical data and can be used to represent things like file systems and organizational charts.
9. Graphs are used to represent networks of nodes and edges, and they are commonly used in applications such as social networks and transportation systems.

◆ Introduction to Lists:

Lists are a commonly used data structure in Python that can store collections of items. A list is an ordered sequence of elements, and each element can be of a different data type, such as numbers, strings, or other objects. Lists can be modified after they are created, which means that you can add, remove, or modify elements as needed.

➤ List Methods:

Lists have a number of built-in methods that allow you to manipulate the elements of a list. Some of the most commonly used list methods include:

1. **append():** Adds an element to the end of a list.
2. **insert():** Adds an element to a specific index in the list.
3. **remove():** Removes the first occurrence of an element in the list.
4. **pop():** Removes and returns the element at a specific index in the list.
5. **sort():** Sorts the elements in the list in ascending order.
6. **reverse():** Reverses the order of the elements in the list.

Here are some examples of how each of the list methods can be used in Python:

- **append():** Adds an element to the end of a list.

```
//@https://codewithcurious.com/  
  
fruits = ['apple', 'banana', 'cherry']  
fruits.append('orange')  
print(fruits) # Output: ['apple', 'banana', 'cherry', 'orange']
```

- **insert()**: Adds an element to a specific index in the list.



```
//@https://codewithcurious.com/
```

```
fruits = ['apple', 'banana', 'cherry']  
fruits.insert(1, 'orange')  
print(fruits) # Output: ['apple', 'orange', 'banana', 'cherry']
```

- **remove()**: Removes the first occurrence of an element in the list.



```
//@https://codewithcurious.com/  
  
fruits = ['apple', 'banana', 'cherry']  
fruits.remove('banana')  
print(fruits) # Output: ['apple', 'cherry']
```

- **pop():** Removes and returns the element at a specific index in the list.



```
//@https://codewithcurious.com/  
  
fruits = ['apple', 'banana', 'cherry']  
popped_fruit = fruits.pop(1)  
print(popped_fruit) # Output: 'banana'  
print(fruits) # Output: ['apple', 'cherry']
```

- **sort():** Sorts the elements in the list in ascending order.

PDF FILE UPLOADED ON TELEGRAM {Link in Bio}



```
//@https://codewithcurious.com/
```

```
numbers = [4, 2, 1, 3, 5]  
numbers.sort()  
print(numbers) # Output: [1, 2, 3, 4, 5]
```

- **reverse():** Reverses the order of the elements in the list.



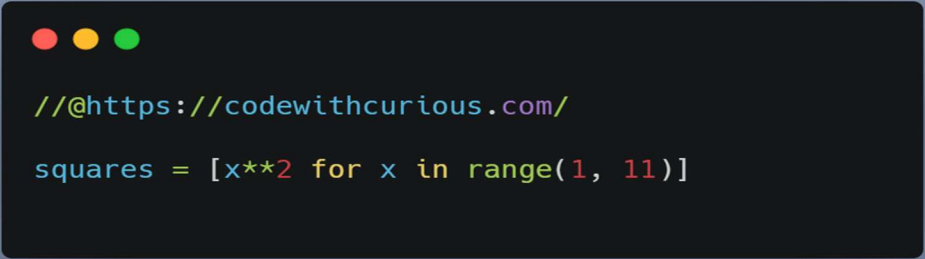
```
//@https://codewithcurious.com/
```

```
fruits = ['apple', 'banana', 'cherry']  
fruits.reverse()  
print(fruits) # Output: ['cherry', 'banana',  
                  'apple']
```

Note that these are just a few examples of how the list methods can be used in Python. There are many other methods available for manipulating lists, so it's worth taking the time to explore the full range of list operations that Python provides.

➤ **List Comprehensions:**

List comprehensions provide a concise way to create lists in Python. They allow you to create a new list by applying an expression to each element of an existing list or other iterable object. Here's an example of a list comprehension that creates a new list containing the squares of the numbers from 1 to 10:



```
//@https://codewithcurious.com/  
squares = [x**2 for x in range(1, 11)]
```

This creates a new list called squares that contains the squares of the numbers from 1 to 10. The expression `x**2` is applied to each element of the range object `range(1, 11)`,

PDF FILE UPLOADED ON TELEGRAM {Link in Bio}

which generates the numbers from 1 to 10. The result is a new list containing the values [1, 4, 9, 16, 25, 36, 49, 64, 81, 100].

List comprehensions can also include conditional expressions, which allow you to filter the elements of an existing list based on some criteria. For example, here's a list comprehension that creates a new list containing only the even numbers from an existing list:

```
//@https://codewithcurious.com/  
  
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
even_numbers = [x for x in numbers if x % 2 == 0]
```

This creates a new list called `even_numbers` that contains only the even numbers from the list `numbers`. The expression `x for x in numbers` is applied to each element of the list, but only the elements that satisfy the condition `x % 2 == 0` are included in the new list. The result is a new list containing the values [2, 4, 6, 8, 10].

Here's another example of a list comprehension that creates a new list containing the lengths of strings in an existing list:

```
//@https://codewithcurious.com/  
  
words = ["apple", "banana", "cherry", "date"]  
lengths = [len(word) for word in words]
```

This creates a new list called `lengths` that contains the lengths of the strings in the list `words`. The expression `len(word)` is applied to each element of the list `words`, which generates the lengths of the strings. The result is a new list containing the values `[5, 6, 6, 4]`.

List comprehensions are a powerful and concise way to create new lists in Python, and they are commonly used in a variety of applications.

◆ Introduction to Tuples :

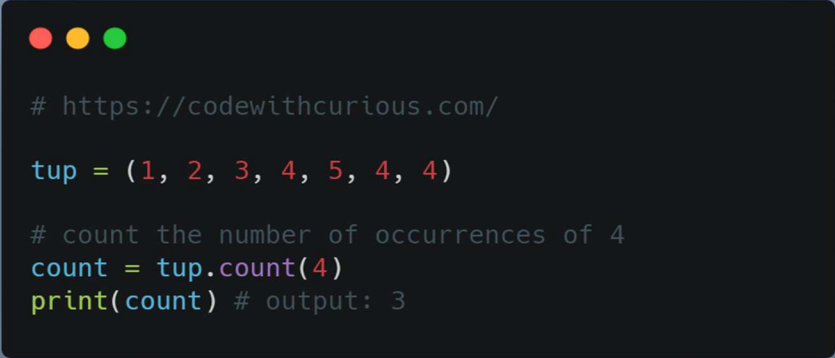
A tuple is a collection of ordered, immutable, and heterogeneous elements enclosed in parentheses `()`. Tuples are similar to lists, but the difference is that tuples cannot be modified once created, whereas lists can be modified. Tuples are commonly used to group related data together.

➤ Tuple Methods with Example:

- **count():**

The count() method is used to count the number of occurrences of a specified element in a tuple.

Example:



```
# https://codewithcurious.com/

tup = (1, 2, 3, 4, 5, 4, 4)

# count the number of occurrences of 4
count = tup.count(4)
print(count) # output: 3
```

- **index():**

The index() method is used to find the index of the first occurrence of a specified element in a tuple.

Example:

```
# https://codewithcurious.com/

# create a tuple
tup = (1, 2, 3, 4, 5, 4, 4)

# find the index of the first occurrence of 4
index = tup.index(4)
print(index) # output: 3
```

- **len():**

The len() method is used to get the length of a tuple.

Example:

```
# https://codewithcurious.com/

# create a tuple
# create a tuple
tup = (1, 2, 3, 4, 5)

# get the length of the tuple
length = len(tup)
print(length) # output: 5
```

- **sorted():**

The sorted() method is used to get a new sorted tuple from an existing tuple.

Example:

```
# https://codewithcurious.com/

# create a tuple
tup = (5, 2, 8, 1, 6)

# get a new sorted tuple
sorted_tup = sorted(tup)
print(sorted_tup) # output: [1, 2, 5, 6, 8]
```

- **min() and max():**

The min() and max() methods are used to get the minimum and maximum value from a tuple.

Example:

```
# https://codewithcurious.com/

# create a tuple
# create a tuple
tup = (5, 2, 8, 1, 6)

# get the minimum and maximum value from the tuple
min_val = min(tup)
max_val = max(tup)
print(min_val) # output: 1
print(max_val) # output: 8
```

- ◆ **Introduction to Sets:**

A set is an unordered collection of unique elements. The sets can be created using the built-in set() function or by enclosing a comma-separated list of elements inside curly braces {}.

PDF FILE UPLOADED ON TELEGRAM {Link in Bio}

Example:

```
● ● ●  
# https://codewithcurious.com/  
my_set = {1, 2, 3, 4, 5}  
print(my_set) # Output: {1, 2, 3, 4, 5}
```

➤ **Set Methods:**

- **add():** Adds an element to the set. If the element already exists in the set, the set remains unchanged.

Example:

```
● ● ●  
# https://codewithcurious.com/  
# Creating a set  
my_set = {1, 2, 3, 4, 5}  
  
# Adding an element to the set  
my_set.add(6)  
  
print(my_set) # Output: {1, 2, 3, 4, 5, 6}  
  
# Trying to add an existing element to the set  
my_set.add(5)  
  
print(my_set) # Output: {1, 2, 3, 4, 5, 6}
```

- **update():** Adds elements to the set from an iterable.

Example:


```
● ● ●  
  
# https://codewithcurious.com/  
# Creating a set  
# Creating a set  
my_set = {1, 2, 3, 4, 5}  
  
# Adding elements to the set using update()  
my_set.update([6, 7, 8])  
  
print(my_set) # Output: {1, 2, 3, 4, 5, 6, 7, 8}
```

- **remove():** Removes an element from the set. If the element does not exist, it raises a `KeyError`.

Example:

```
● ● ●  
  
# https://codewithcurious.com/  
# Creating a set  
my_set = {1, 2, 3, 4, 5}  
  
# Removing an element from the set using remove()  
my_set.remove(3)  
  
print(my_set) # Output: {1, 2, 4, 5}  
  
# Trying to remove a non-existent element from the  
# set  
my_set.remove(3)
```

- **pop():** Removes and returns an arbitrary element from the set. If the set is empty, it raises a `KeyError`.

Example:

```

# https://codewithcurious.com/
# Creating a set
my_set = {1, 2, 3, 4, 5}

# Removing an element from the set using remove()
my_set.remove(3)

print(my_set) # Output: {1, 2, 4, 5}

# Trying to remove a non-existent element from the set
my_set.remove(3)

```

- **clear()** method in Python is used to remove all elements from a set. Here's an example:

```

# https://codewithcurious.com/
# create a set
fruits = {'apple', 'banana', 'orange', 'mango'}

# print the set
print(fruits)

# remove all elements from the set
fruits.clear()

# print the set after clearing it
print(fruits)

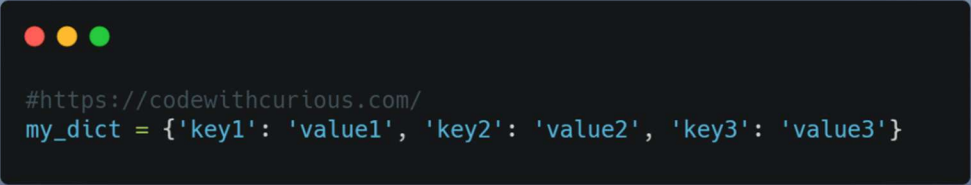
```

◆ Introduction to Dictionaries:

Dictionaries are a type of collection in Python that allows you to store and retrieve values based on keys. Each key in a dictionary is unique and associated

PDF FILE UPLOADED ON TELEGRAM {Link in Bio}

with a value. Dictionaries are created using curly braces {} and are separated by commas. The syntax for creating a dictionary is as follows:

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains two lines of Python code: a comment line starting with # and a dictionary assignment line.

```
#https://codewithcurious.com/  
my_dict = {'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}
```

In the above example, `my_dict` is a dictionary with three key-value pairs. The keys are 'key1', 'key2', and 'key3', and their corresponding values are 'value1', 'value2', and 'value3', respectively.

➤ Dictionary Methods:

Dictionaries have several built-in methods that you can use to manipulate and retrieve data from them. Here are some of the most commonly used dictionary methods:

- **`dict.get(key, default=None)`:** Returns the value associated with the specified key. If the key is not found, it returns the specified default value.
- **`dict.keys()`:** Returns a list of all the keys in the dictionary.
- **`dict.values()`:** Returns a list of all the values in the dictionary.
- **`dict.items()`:** Returns a list of tuples containing key-value pairs.
- **`dict.update(other_dict)`:** Adds the key-value pairs from another dictionary to the current dictionary.
- **`dict.pop(key, default=None)`:** Removes the key-value pair associated with the specified key. If the key is not found, it returns the specified default value.
- **`dict.clear()`:** Removes all the key-value pairs from the dictionary.
- **`len(dict)`:** Returns the number of key-value pairs in the dictionary.

These are just a few of the methods available for working with dictionaries in Python. By using these methods, you can easily manipulate and retrieve data from your dictionaries.

example of a dictionary and how to use some of the methods mentioned:

PDF FILE UPLOADED ON TELEGRAM {Link in Bio}

```

#https://codewithcurious.com/
my_dict = {'apple': 3, 'banana': 2, 'orange': 5}

# Accessing a value with a key
print(my_dict['apple']) # Output: 3

# Using the get() method to access a value with a key
print(my_dict.get('banana')) # Output: 2

# Getting a list of all the keys
print(my_dict.keys()) # Output: ['apple', 'banana', 'orange']

# Getting a list of all the values
print(my_dict.values()) # Output: [3, 2, 5]

# Getting a list of all the key-value pairs
print(my_dict.items()) # Output: [('apple', 3), ('banana', 2), ('orange', 5)]

# Updating the dictionary with another dictionary
new_dict = {'grape': 1, 'kiwi': 4}
my_dict.update(new_dict)
print(my_dict) # Output: {'apple': 3, 'banana': 2, 'orange': 5, 'grape': 1, 'kiwi': 4}

# Removing a key-value pair with pop()
my_dict.pop('orange')
print(my_dict) # Output: {'apple': 3, 'banana': 2, 'grape': 1, 'kiwi': 4}

# Removing all the key-value pairs with clear()
my_dict.clear()
print(my_dict) # Output: {}

# Getting the length of the dictionary
print(len(my_dict)) # Output: 0

```

● Introduction to Arrays:

In Python, arrays are a type of data structure used to store homogeneous items. They can be more efficient than lists for certain types of operations, especially when dealing with large amounts of numerical data.

PDF FILE UPLOADED ON TELEGRAM {Link in Bio}

Arrays are created using the `array()` function from the built-in `array` module. The syntax for creating an array is as follows:

```
#https://codewithcurious.com/  
  
import array  
my_array = array.array('typecode', [item1, item2, item3, ...])
```

In the above code, `typecode` specifies the type of data that the array will hold, and `[item1, item2, item3, ...]` is a list of items to be stored in the array.

➤ Array Methods:

Arrays have several built-in methods that you can use to manipulate and retrieve data from them. Here are some of the most commonly used array methods:

- **`array.append(item)`:** Adds an item to the end of the array.
- **`array.extend(iterable)`:** Adds the items from an iterable to the end of the array.
- **`array.insert(index, item)`:** Inserts an item at the specified index.
- **`array.remove(item)`:** Removes the first occurrence of an item from the array.
- **`array.pop([index])`:** Removes and returns the item at the specified index. If no index is specified, it removes and returns the last item.
- **`array.index(item)`:** Returns the index of the first occurrence of an item in the array.
- **`array.count(item)`:** Returns the number of occurrences of an item in the array.

PDF FILE UPLOADED ON TELEGRAM {Link in Bio}

- **array.reverse()** : Reverses the order of the items in the array.
- **array.tobytes()** : Returns the array as a bytes object.
- **array.frombytes(bytes)** : Creates a new array from a bytes object.

These are just a few of the methods available for working with arrays in Python. By using these methods, you can easily manipulate and retrieve data from your arrays.

here's an example of creating and manipulating an array using the methods mentioned:

```
#https://codewithcurious.com/

import array

# Create an array of integers
my_array = array.array('i', [1, 2, 3, 4, 5])

# Append an item to the end of the array
my_array.append(6)
print(my_array) # Output: array('i', [1, 2, 3, 4, 5, 6])

# Extend the array with items from a list
my_array.extend([7, 8, 9])
print(my_array) # Output: array('i', [1, 2, 3, 4, 5, 6, 7, 8, 9])

# Insert an item at a specific index
my_array.insert(3, 10)
print(my_array) # Output: array('i', [1, 2, 3, 10, 4, 5, 6, 7, 8, 9])
```

```

#https://codewithcurious.com/

import array

# Remove an item from the array
my_array.remove(5)
print(my_array) # Output: array('i', [1, 2, 3, 10, 4, 6, 7, 8, 9])

# Pop an item from the array
popped_item = my_array.pop()
print(popped_item) # Output: 9
print(my_array) # Output: array('i', [1, 2, 3, 10, 4, 6, 7, 8])

# Find the index of an item
index = my_array.index(6)
print(index) # Output: 5

# Count the number of occurrences of an item
count = my_array.count(3)
print(count) # Output: 1

# Reverse the order of the items in the array
my_array.reverse()
print(my_array) # Output: array('i', [8, 7, 6, 4, 10, 3, 2, 1])

# Convert the array to a bytes object
bytes_array = my_array.tobytes()
print(bytes_array) # Output:
b'\x08\x00\x00\x00\x07\x00\x00\x00\x06\x00\x00\x00\x04\x00\x00\x00\n\x00\x00\x00\x03\x00\x00\x00\x02\x00\x00\x00\x01\x00\x00\x00'

# Create a new array from a bytes object
new_array = array.array('i')
new_array.frombytes(bytes_array)
print(new_array) # Output: array('i', [8, 7, 6, 4, 10, 3, 2, 1])

```

Code

This program creates an array of integers using the `array()` function from the `array` module. It then uses several array methods such as `append()`, `extend()`, `insert()`, `remove()`, `pop()`, `index()`, `count()`, `reverse()`, `tobytes()`, and `frombytes()` to manipulate and retrieve data from the array. The output of each operation is printed to the console.

PDF FILE UPLOADED ON TELEGRAM {Link in Bio}

◆ Introduction to Queues:

A queue is a linear data structure in which items can be added to one end of the queue and removed from the other end of the queue. This is known as a first-in, first-out (FIFO) data structure because the first item added to the queue is the first item removed from the queue.

➤ Queue Methods:

Python has an inbuilt module called queue that provides a Queue class that implements a queue data structure. The Queue class provides several methods for adding, removing, and manipulating items in the queue:

- **Queue():** - creates a new queue object.
- **put(item):** - adds an item to the end of the queue.
- **get():** - removes and returns the first item from the queue.
- **empty():** - returns True if the queue is empty, otherwise returns False.
- **qsize():** - returns the number of items in the queue.
- **task_done():** - signals that a task previously pulled from the queue has been completed.
- **join():** - blocks until all tasks have been completed and marked as done.

Here's an example of how to use the queue module in Python to create and manipulate a queue:


```
#https://codewithcurious.com/

import queue

# Create a new queue object
my_queue = queue.Queue()

# Add items to the queue
my_queue.put('apple')
my_queue.put('banana')
my_queue.put('cherry')

# Check if the queue is empty
print(my_queue.empty()) # Output: False

# Get and remove the first item from the queue
first_item = my_queue.get()
print(first_item) # Output: 'apple'

# Get the number of items in the queue
num_items = my_queue.qsize()
print(num_items) # Output: 2

# Mark a task as done
my_queue.task_done()

# Block until all tasks are done
my_queue.join()
```

In this example, we first import the queue module and create a new queue object using the **Queue()** method. We then add items to the queue using the **put()** method, and remove the first item from the queue using the **get()** method. We also use the **empty()** and **qsize()** methods to check the status of the queue, and the **task_done()** and **join()** methods to signal when tasks have been completed.

◆ Introduction to Stacks :

A stack is a linear data structure in which items can be added or removed only from one end of the stack, which is called the top. This is known as a last-in, first-out (LIFO) data structure because the last item added to the stack is the first item removed from the stack.

A stack is a collection of elements that supports two main operations: push, which adds an element to the collection, and pop, which removes the most recently added element. Stacks are used in a variety of applications, including parsing expressions, evaluating arithmetic expressions, and depth-first search algorithms. In Python, you can implement a stack using a list, which provides the necessary push and pop methods.

➤ Stack Methods:

Python does not have a built-in stack data structure, but you can use a list to implement a stack. A list provides several methods that can be used to implement the stack behavior:

- **append(item):** - adds an item to the top of the stack.
- **pop():** - removes and returns the item at the top of the stack.
- **[-1]:** - returns the item at the top of the stack without removing it.
- **len():** - returns the number of items in the stack.

Here's an example of how to use a list to implement a stack in Python:

```
#https://codewithcurious.com/
# Create an empty stack
my_stack = []

# Add items to the top of the stack
my_stack.append('apple')
my_stack.append('banana')
my_stack.append('cherry')

# Check the top item of the stack
print(my_stack[-1]) # Output: 'cherry'

# Remove and return the top item of the stack
top_item = my_stack.pop()
print(top_item) # Output: 'cherry'

# Check the number of items in the stack
num_items = len(my_stack)
print(num_items) # Output: 2
```

In this example, we use an empty list to create a stack. We add items to the top of the stack using the `append()` method, and remove the top item from the stack using the `pop()` method. We also use the `[-1]` index to access the top item of the stack without removing it, and the `len()` method to check the number of items in the stack.

◆ Introduction to Linked Lists:

A linked list is a data structure in which each element, called a node, contains a value and a reference to the next node in the list. Unlike arrays or lists, linked lists do not store their elements contiguously in memory, but rather use pointers to link the elements together.

➤ Linked List Methods:

Linked lists provide several methods for accessing and manipulating their elements, including:

- **append(value):** - adds a new node with the specified value to the end of the linked list.
- **insert(value, index):** - adds a new node with the specified value at the specified index in the linked list.
- **remove(value):** - removes the first node in the linked list that contains the specified value.
- **pop(index):** - removes and returns the node at the specified index in the linked list.
- **get(index):** - returns the value of the node at the specified index in the linked list.

Here's an example of how to create a linked list in Python and use some of these methods:

PDF FILE UPLOADED ON TELEGRAM {Link in Bio}

CodeWithCurious.com

```
#https://codewithcurious.com/
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, value):
        new_node = Node(value)
        if not self.head:
            self.head = new_node
        else:
            current_node = self.head
            while current_node.next:
                current_node = current_node.next
            current_node.next = new_node

    def insert(self, value, index):
        new_node = Node(value)
        if index == 0:
            new_node.next = self.head
            self.head = new_node
        else:
            current_node = self.head
            for i in range(index - 1):
                current_node = current_node.next
            new_node.next = current_node.next
            current_node.next = new_node

    def remove(self, value):
        if not self.head:
            return
        if self.head.value == value:
            self.head = self.head.next
            return
        current_node = self.head
        while current_node.next and current_node.next.value != value:
            current_node = current_node.next
        if current_node.next and current_node.next.value == value:
            current_node.next = current_node.next.next
```

CodeWithCurious.com

```

def pop(self, index):
    if not self.head:
        return None
    if index == 0:
        node = self.head
        self.head = self.head.next
        return node.value
    current_node = self.head
    for i in range(index - 1):
        if not current_node.next:
            return None
        current_node = current_node.next
    if not current_node.next:
        return None
    node = current_node.next
    current_node.next = current_node.next.next
    return node.value

def get(self, index):
    current_node = self.head
    for i in range(index):
        if not current_node:
            return None
        current_node = current_node.next
    return current_node.value

# Create a new linked list
my_list = LinkedList()

# Add some values to the list
my_list.append(1)
my_list.append(2)
my_list.append(3)

# Insert a value at index 1
my_list.insert(4, 1)

# Remove a value from the list
my_list.remove(2)

# Pop a value from the list
popped_value = my_list.pop(1)
print(popped_value)

# Get a value from the list
value = my_list.get(1)
print(value)

```


In this example, we define a `Node` class to represent the nodes in the linked list, and a **`LinkedList`** class to represent the list itself. We then define several methods for the `LinkedList` class, including **`append`**, **`insert`**, **`remove`**, **`pop`**, and **`get`**, which we can use to **`add`**, **`remove`**, and access elements in the linked list.

We create a new linked list object, `my_list`, and add some values to it using the `append` method. We then insert a new value at index 1 using the `insert` method, remove a value from the list using the `remove` method, and **`pop`** a value from the list using the **`pop`** method.

Finally, we use the `get` method to retrieve a value from the list at a specified index, and print the value to the console. This is just a simple example, and there are many other methods and variations on linked lists that can be used for different applications.

◆ Introduction to Trees:

A tree is a collection of nodes that are linked together in a hierarchical structure. Each node in a tree has a parent node (except for the root node) and zero or more child nodes. The root node is the topmost node in the tree, and has no parent. Trees are commonly used to represent hierarchical data, such as file systems, organizational charts, and the HTML DOM.

➤ Tree Traversal Methods :

Traversal refers to the process of visiting each node in a tree. There are several ways to traverse a tree, including depth-first traversal and breadth-first traversal. In depth-first traversal, the nodes are visited starting at the root node and moving down the tree, visiting each child node before moving on to the next sibling node. There are three types of depth-first traversal: pre-order traversal, in-order traversal, and post-order traversal.

In pre-order traversal, the current node is visited first, then the left child, and then the right child. In in-order traversal, the left child is visited first, then the current node, and then the right child. In post-order traversal, the left child is visited first, then the right child, and then the current node.

In breadth-first traversal, the nodes are visited starting at the root node and moving level by level through the tree, visiting all nodes at each level before

PDF FILE UPLOADED ON TELEGRAM {Link in Bio}

moving on to the next level. This is often done using a queue data structure to keep track of the nodes to be visited.

Tree traversal methods are important for many applications, such as searching a tree for a particular node or calculating the depth of a tree.

- In-order traversal is a type of depth-first traversal of a binary tree, in which the nodes are visited in the following order:

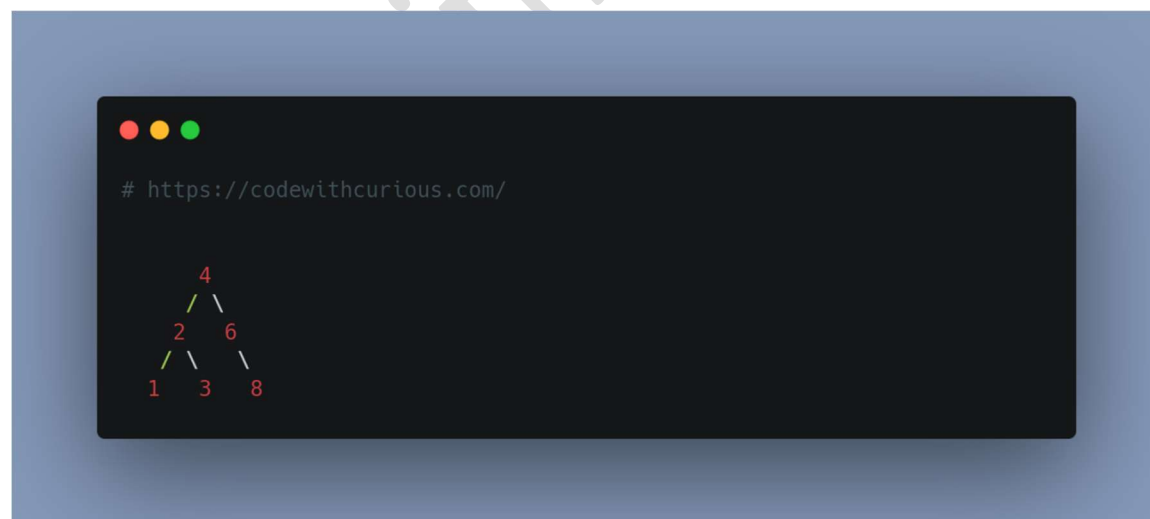
Visit the left subtree.

Visit the root node.

Visit the right subtree.

This means that in-order traversal visits the nodes in ascending order. For example, if we have a binary search tree with the following nodes:

here's an example of a simple binary tree in Python and how to traverse it using depth-first traversal:



The in-order traversal would visit the nodes in the following order: 1, 2, 3, 4, 6, 8.

In the case of a binary tree that is not a binary search tree, the order of visited nodes will not necessarily be in ascending or descending order. Rather, in-order traversal simply visits the left subtree before the root, and the right subtree after the root, for each node in the tree.

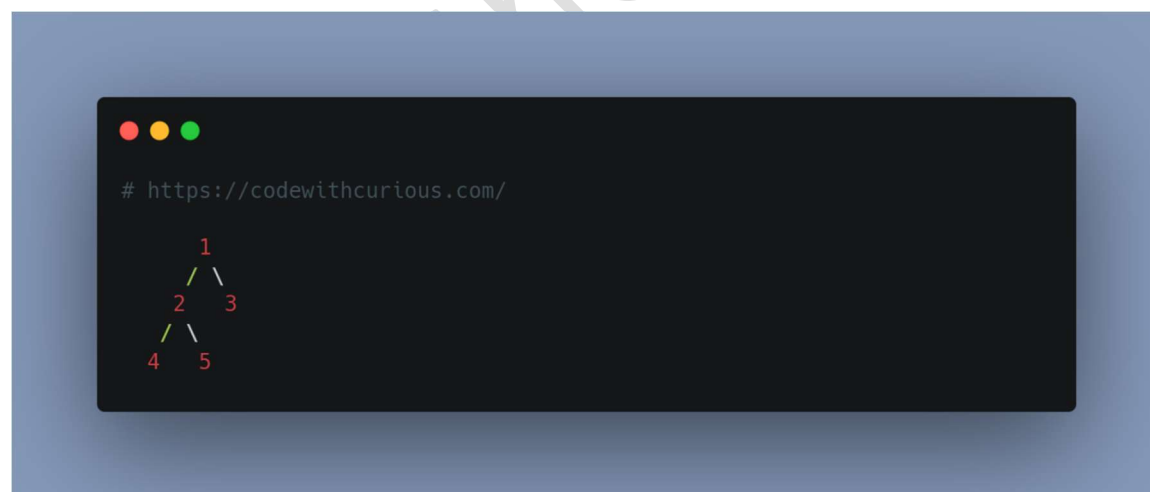
- Pre-order traversal is a type of depth-first traversal of a binary tree, in which the nodes are visited in the following order:

Visit the root node.

Visit the left subtree.

Visit the right subtree.

This means that pre-order traversal visits the root node first, followed by its left subtree, and then its right subtree. For example, if we have a binary tree with the following nodes:



The pre-order traversal would visit the nodes in the following order: 1, 2, 4, 5, 3.

Pre-order traversal can be useful for creating a copy of the tree, as it visits the nodes in the order in which they would be visited if the tree were being copied

PDF FILE UPLOADED ON TELEGRAM {Link in Bio}

recursively. It can also be used to evaluate expressions in a parse tree, as it visits operators before their operands.

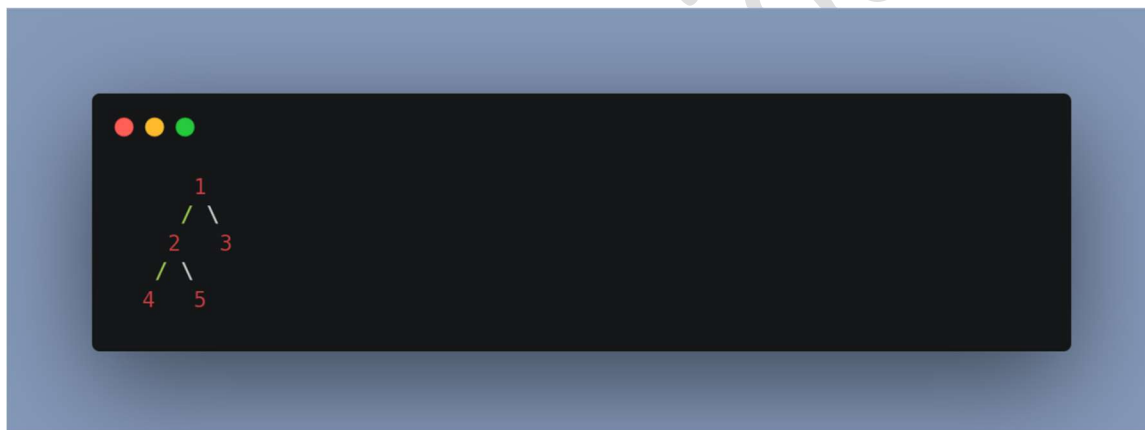
- Post-order traversal is a type of depth-first traversal of a binary tree, in which the nodes are visited in the following order:

Visit the left subtree.

Visit the right subtree.

Visit the root node.

This means that post-order traversal visits the left subtree and right subtree before visiting the root node. For example, if we have a binary tree with the following nodes:



The post-order traversal would visit the nodes in the following order: 4, 5, 2, 3, 1.

Post-order traversal can be useful for deleting a tree, as it visits the leaves of the tree first before moving on to the root. It can also be used to compute the height of the tree or to evaluate the expressions in a parse tree, as it visits the operands before the operators.

here's an example of a simple binary tree in Python and how to traverse it using depth-first traversal:

```
# https://codewithcurious.com/

class Node:
    def __init__(self, data):
        self.left = None
        self.right = None
        self.data = data

    def inorder_traversal(node):
        if node:
            inorder_traversal(node.left)
            print(node.data)
            inorder_traversal(node.right)

    def preorder_traversal(node):
        if node:
            print(node.data)
            preorder_traversal(node.left)
            preorder_traversal(node.right)

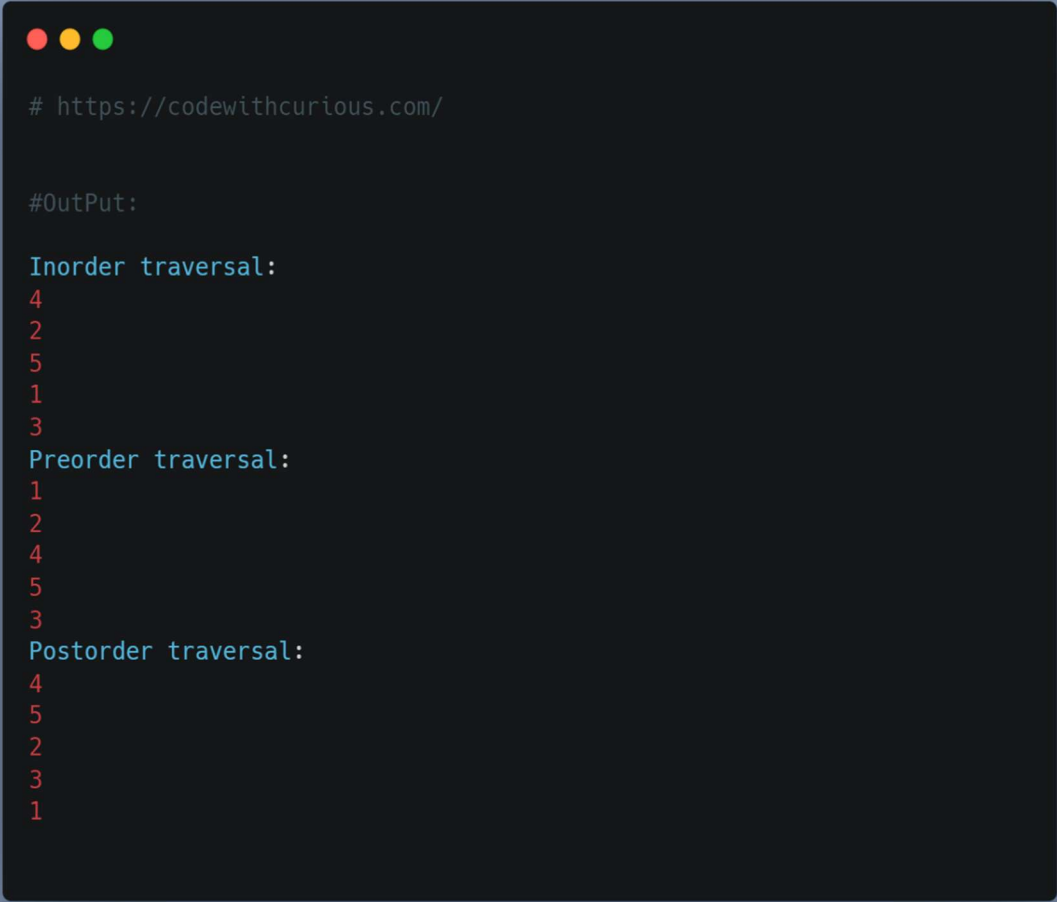
    def postorder_traversal(node):
        if node:
            postorder_traversal(node.left)
            postorder_traversal(node.right)
            print(node.data)

# create the binary tree
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)

# traverse the tree using depth-first traversal
print("Inorder traversal:")
inorder_traversal(root)
print("Preorder traversal:")
preorder_traversal(root)
print("Postorder traversal:")
postorder_traversal(root)
```

In this example, we define a **Node** class to represent a node in the binary tree. Each node has a **left** and **right child**, and a **data value**. We then define three traversal methods: **inorder_traversal**, **preorder_traversal**, and **postorder_traversal**, which implement the three types of depth-first traversal described earlier.

We create a binary tree with five nodes, and then traverse it using each of the three traversal methods. The output of the program will be:



```
# https://codewithcurious.com/

#OutPut:

Inorder traversal:
4
2
5
1
3
Preorder traversal:
1
2
4
5
3
Postorder traversal:
4
5
2
3
1
```


As you can see, the order in which the nodes are visited depends on the traversal method used. In-order traversal visits the nodes in ascending order, while pre-order traversal visits the root node first, and post-order traversal visits the root node last.

◆ Introduction to Graphs :

Graphs are a collection of nodes (also called vertices) and edges that connect these nodes. They are used to model relationships or connections between objects, such as social networks or transportation systems.

Graphs can be represented in different ways, such as adjacency matrix and adjacency list. An adjacency matrix is a two-dimensional array where the presence of an edge between two vertices is denoted by a 1, while an adjacency list is a list of all the vertices in the graph along with a list of their adjacent vertices.

➤ Graph Traversal Methods :

PDF FILE UPLOADED ON TELEGRAM {Link in Bio}

Graph traversal is the process of visiting all the nodes in a graph in a systematic way. There are two main traversal methods for graphs:

- **Breadth-first traversal:** In this method, the graph is traversed by exploring all the nodes at a given level before moving on to the next level. This is often used to find the shortest path between two nodes.
- **Depth-first traversal:** In this method, the graph is traversed by exploring as far as possible along each branch before backtracking. This is often used to find all possible paths between two nodes.

Both breadth-first and depth-first traversal can be implemented using recursive or iterative algorithms. Other graph algorithms include Dijkstra's shortest path algorithm and Kruskal's minimum spanning tree algorithm, which are used to find the shortest path and minimum spanning tree of a graph, respectively.

here's a Python example of a graph represented as an adjacency list, and a breadth-first traversal algorithm to visit all the nodes:

```

# https://codewithcurious.com/

class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v):
        if u not in self.graph:
            self.graph[u] = []
        self.graph[u].append(v)

def breadth_first_traversal(graph, start):
    visited = set()
    queue = [start]
    while queue:
        node = queue.pop(0)
        if node not in visited:
            visited.add(node)
            print(node, end=' ')
            for neighbor in graph.get(node, []):
                if neighbor not in visited:
                    queue.append(neighbor)

# example usage
g = Graph()
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 2)
g.add_edge(2, 0)
g.add_edge(2, 3)
g.add_edge(3, 3)

print("Breadth-first traversal starting from node 2:")
breadth_first_traversal(g.graph, 2)

```

This code defines a Graph class that uses an adjacency list to represent the graph, and a breadth_first_traversal function that takes a graph and a starting node and visits all the nodes in the graph using breadth-first traversal.

PDF FILE UPLOADED ON TELEGRAM {Link in Bio}

In this example, the graph has 4 nodes (0, 1, 2, and 3) and 6 edges connecting them. The breadth-first traversal starting from node 2 visits the nodes in the order 2, 0, 3, 1.