**Data Structures**

Python offers a variety of built-in data structures to store, organize, and manipulate data efficiently. Here are some of the most used data structures in Python:

1. Lists:

- Lists are ordered collections of items that can be of mixed data types. They are defined using square brackets and support indexing, slicing, and various methods for adding, removing, and modifying elements.

```python

my\_list = [1, 'apple', 3.14, [4, 5]]

```

2. Tuples:

- Tuples are like lists but are immutable, meaning their elements cannot be changed after creation. They are defined using parentheses and are often used to represent fixed collections of items.

```python

my\_tuple = (1, 'banana', 2.71)

```

3. Sets:

- Sets are unordered collections of unique elements. They are defined using curly braces or the `set()` constructor and are useful for performing set operations like union, intersection, and difference.

```python

my\_set = {1, 2, 3}

```

4. Dictionaries:

- Dictionaries are collections of key-value pairs, where each key is unique. They are defined using curly braces and colons and allow you to efficiently retrieve values by their associated keys.

```python

my\_dict = {'name': 'Alice', 'age': 30, 'city': 'New York'}

```

5. Strings:

- Strings are sequences of characters and are enclosed in single, double, or triple quotes. Python provides many string manipulation methods and supports various string operations.

```python

my\_string = "Hello, Python!"

```

6. Arrays (NumPy):

- NumPy is a popular library for numerical computing in Python. It provides multidimensional arrays called `ndarrays` that are highly efficient for mathematical and scientific computations.

```python

import numpy as np

my\_array = np.array([1, 2, 3])

```

7. Stacks and Queues (collections.deque):

- Python's `collections` module includes the `deque` data structure, which can be used to implement both stacks and queues efficiently.

```python

from collections import deque

my\_stack = deque()

my\_queue = deque()

```

8. Linked Lists (custom implementations):

- While Python doesn't provide a built-in linked list data structure, you can create your own custom linked lists using classes and objects.

```python

class Node:

def \_\_init\_\_(self, data):

self.data = data

self.next = None

```

9. Dictionaries with Default Values (collections.defaultdict):

- The `defaultdict` is a specialized dictionary provided by the `collections` module that allows you to set a default value for missing keys.

```python

from collections import defaultdict

my\_default\_dict = defaultdict(int)

```

10. Heaps (heapq):

- Python's `heapq` module provides functions for creating and manipulating min-heaps. Min-heaps are often used for tasks like finding the smallest element in a collection.

```python

import heapq

my\_heap = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]

heapq.heapify(my\_heap)

```

In Python, a list is a fundamental and versatile data structure used to store a collection of items. Lists are ordered, mutable (modifiable), and allow for duplicates. You can create lists containing various data types, including numbers, strings, and even other lists. Lists are denoted by square brackets, and elements within a list are separated by commas. Here's an overview of Python lists and the types of lists:

**Basic List Operations:**

* Creating a list:

my\_list = [1, 2, 3, 4, 5]

* Accessing elements by index:

first\_element = my\_list[0] # Access the first element (index 0)

* Modifying elements:

my\_list[2] = 10 # Change the third element to 10

* List concatenation:

new\_list = my\_list + [6, 7, 8]

* List repetition:

repeated\_list = my\_list \* 3 # Creates a new list with three copies of the original list

* Finding the length of a list:

length = len(my\_list) # Returns the number of elements in the list

**Common List Operations:**

* Adding elements to the end of a list:

my\_list.append(6) # Appends 6 to the end of the list

* Inserting elements at a specific position:

my\_list.insert(2, 7) # Inserts 7 at index 2, shifting the rest of the elements

* Removing elements:

my\_list.remove(4) # Removes the first occurrence of 4 from the list

* Popping elements:

popped\_element = my\_list.pop() # Removes and returns the last element

* Finding the index of an element:

index = my\_list.index(3) # Returns the index of the first occurrence of 3

* Counting occurrences:

count = my\_list.count(2) # Returns the number of times 2 appears in the list

* Sorting a list:

my\_list.sort() # Sorts the list in ascending order

**Types of Lists:**

1. **Numeric Lists**: Contain numeric values like integers or floating-point numbers.

numbers = [1, 2, 3, 4, 5]

1. **String Lists**: Contain strings as elements.

names = ["Alice", "Bob", "Charlie"]

1. **Nested Lists**: Lists can contain other lists as elements, creating a nested or multi-dimensional structure.

matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

1. **Mixed-Type Lists**: Python allows lists to hold a mixture of different data types.

mixed = [1, "apple", 3.14, [4, 5, 6]]

1. **Empty Lists**: Lists with no elements can be created.

empty\_list = []

**classes**

In Python, a class is a blueprint for creating objects. Objects are instances of classes, and classes define the properties (attributes) and behaviors (methods) of those objects. Here's a basic example of a class in Python:

```python

class Dog:

# Class variable

species = "Canis familiaris"

# Constructor method

def \_\_init\_\_(self, name, age):

# Instance variables

self.name = name

self.age = age

# Instance method

def bark(self):

return "Woof!"

# Creating an instance of the Dog class

my\_dog = Dog(name="Buddy", age=3)

# Accessing attributes

print(f"{my\_dog.name} is {my\_dog.age} years old.")

# Calling a method

print(my\_dog.bark())

```

Let's break down the components:

- \*\*Class:\*\* The `Dog` class is defined using the `class` keyword.

- \*\*Class Variables:\*\* The `species` variable is a class variable, shared by all instances of the class.

- \*\*Constructor (`\_\_init\_\_`):\*\* The `\_\_init\_\_` method is a special method called the constructor. It is executed when a new object is created and is used to initialize the object's attributes.

- \*\*Instance Variables:\*\* `self.name` and `self.age` are instance variables, unique to each instance of the class.

- \*\*Instance Method (`bark`):\*\* The `bark` method is a function defined within the class. It operates on the instance of the class and can access its attributes.

- \*\*Creating an Instance (`my\_dog`):\*\* An instance of the `Dog` class is created, and values are passed to the constructor to initialize its attributes.

- \*\*Accessing Attributes and Calling Methods:\*\* We can access the attributes of the instance using dot notation (`my\_dog.name`) and call methods (`my\_dog.bark()`).

This is a basic example, and classes in Python can become more complex, with inheritance, encapsulation, and other features. Classes are fundamental to object-oriented programming in Python, providing a way to structure and organize code in a modular and reusable manner.