Data Structure

Data structures are ways to organize and store data in a computer so that we can use it efficiently.

Common data structures in Python:

1. List – an ordered collection (like a shopping list)
2. Dictionary – a collection of key-value pairs (like a contact list)
3. Set – an unordered collection of unique items
4. Tuple – an ordered, unchangeable collection

List

A list in Python is a mutable, ordered collection that can store multiple data types, including numbers, strings, and even other lists. Lists are defined using square brackets [].

Use a List (list) When:

1. Data needs to be modified (add, remove, or change elements).
2. Ordering matters and may change over time.
3. Operations like sorting, appending, or filtering are needed.
4. Data is expected to grow (tuples are fixed size).

In [91]:
```
# Empty List
empty_list = []

# List with integers
numbers = [1, 2, 3, 4, 5]

# List with mixed data types
mixed = [10, "Python", 3.14, True]

# List containing another list (Nested List)
nested = [[1, 2], [3, 4], [5, 6]]

# List with duplicate values
duplicates = [1, 2, 2, 3, 4, 4, 5]
```

In [89]:
```
my_list = ["a", "b", "c", "d"] # List
print(my_list[0])  # Output: 'a'
print(my_list[-1]) # Output: 'd' (Last element)
```

```
a
d
```

In [90]:
```
print(my_list[1:3])  # Output: ['b', 'c'] (From index 1 to 2)
print(my_list[::-1]) # Output: ['d', 'c', 'b', 'a'] (Reversed list)
```

```
['b', 'c']
['d', 'c', 'b', 'a']
```

Python List Methods

Adding Elements

In [1]:
```
# append() - Add an item at the end
numbers = [1, 2, 3]
numbers.append(4)
print(numbers)  # Output: [1, 2, 3, 4]
```

```
[1, 2, 3, 4]
```

In [2]:
```
# append() - Add an item at the end
numbers = [1, 2, 3]
numbers.append([7,8])
print(numbers)
```

```
[1, 2, 3, [7, 8]]
```

In [94]:
```
# extend() - Merge two lists
list1 = [1, 2, 3]
list2 = [4, 5, 6]
list1.extend(list2)
print(list1)  # Output: [1, 2, 3, 4, 5, 6]
```

```
[1, 2, 3, 4, 5, 6]
```

In [19]:
```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
a = list1.extend(list2)
```

In [20]:
```
a
```

In [95]:
```
# insert() - Insert at a specific index
fruits = ["apple", "banana"]
fruits.insert(1, "orange")
print(fruits)  # Output: ['apple', 'orange', 'banana']
```

```
['apple', 'orange', 'banana']
```

Removing Elements

In [5]:
```
# remove() - Remove first occurance of the item
numbers = [1, 2, 3,3,5,3]
numbers.remove(3)
print(numbers)  # Output: []
```

```
[1, 2, 3, 5, 3]
```

In [102…
```
# pop() - Remove and returns item at index i (or last item if index not provided)
numbers = [1, 2, 3]
```

```
removed_item = numbers.pop(1)
print(removed_item)  # Output: 2
print(numbers)  # Output: [1, 3]
```

```
2
[1, 3]
```

In [6]:
```
# pop() - Remove and returns item at index i (or last item if index not provided)
numbers = [1, 2, 3]
removed_item = numbers.pop()
print(removed_item)  # Output: 2
print(numbers)  # Output: [1, 3]
```

```
3
[1, 2]
```

In [101…
```
# clear() - Remove all elements
numbers = [1, 2, 3]
numbers.clear()
print(numbers)  # Output: []
```

```
[]
```

Searching and Counting

In [97]:
```
# index() - Find the position of an item
fruits = ["apple", "banana", "cherry"]
print(fruits.index("banana"))  # Output: 1
```

```
1
```

In [21]:
```
a = [1,2,3,4,5,2,3]
a.index(2)
```

Out[21]: 1

In [98]:
```
# count() - Count occurrences
numbers = [1, 2, 2, 3, 2]
print(numbers.count(2))  # Output: 3
```

```
3
```

Reordering

In [103…
```
# sort() - Sort a list
numbers = [3, 1, 4, 2]
numbers.sort()
print(numbers)  # Output: [1, 2, 3, 4]
```

```
[1, 2, 3, 4]
```

In [9]:
```
# reverse() - Reverse the list
numbers = [3, 1, 4, 2]
numbers.sort(reverse=True)
print(numbers)  # Output: [3, 2, 1]
```

```
[4, 3, 2, 1]
```

In [22]:
```
# reverse() - Reverse the list
numbers = [1, 2, 3]
numbers.reverse()
print(numbers)  # Output: [3, 2, 1]
```

```
[3, 2, 1]
```

Copying

In [105…
```
# copy() - Copy a list
numbers = [1, 2, 3]
new_list = numbers.copy()
print(new_list)  # Output: [1, 2, 3]
```

```
[1, 2, 3]
```

Dictionary

A dictionary in Python is an unordered collection of key-value pairs.

1. Each key is unique and is associated with a value.
2. Dictionaries are defined using curly braces {} with the key-value pairs separated by a colon :.
3. Mutable – You can change, add, or remove items.
4. Keys are Unique – No two keys can be the same.

In [1]:
```
# Basic Dictionary
student = {
    "name": "John",
    "age": 20,
    "course": "Computer Science"
}
print(student)  # Output: {'name': 'John', 'age': 20, 'course': 'Computer Scienc
```

```
{'name': 'John', 'age': 20, 'course': 'Computer Science'}
```

In [2]:
```
# Dictionary with Different Data Types
person = {
    "name": "Alice",
    "age": 25,
    "is_student": False,
    "marks": [80, 90, 85]
}
print(person)  # Output: {'name': 'Alice', 'age': 25, 'is_student': False, 'mark
```

```
{'name': 'Alice', 'age': 25, 'is_student': False, 'marks': [80, 90, 85]}
```

Access & Information

In [3]:
```
# copy() - Create a shallow copy
original = {"name": "Alice", "age": 30}
```

```
copied_dict = original.copy()
print(copied_dict)  # Output: {'name': 'Alice', 'age': 30}
```

```
{'name': 'Alice', 'age': 30}
```

In [4]:
```
# items() – Get all key-value pairs

student = {"name": "John", "age": 20}
print(student.items())  # Output: dict_items([('name', 'John'), ('age', 20)])
```

```
dict_items([('name', 'John'), ('age', 20)])
```

In [5]:
```
# keys() – Get all keys
student = {"name": "John", "age": 20}
print(student.keys())  # Output: dict_keys(['name', 'age'])
```

```
dict_keys(['name', 'age'])
```

In [8]:
```
# values() – Returns a view of all values
student = {"name": "John", "age": 20}
print(student.values())  # Output: dict_values(['John', 20])
```

```
dict_values(['John', 20])
```

In [121…
```
# get() – Access values safely
# Returns value for key, or default if not found
student = {"name": "John", "age": 20, "course": "Python"}
print(student.get("course"))  # Output: Python
print(student.get("address", "Not Available"))  # Output: Not Available
```

```
Python
Not Available
```

Add / Update Elements

In [10]:
```
person = {'name': 'Alice', 'age': 25}

# Add a new key-value pair
person.update({'city': 'Dhaka'})

# Update an existing key
person.update({'age': 26})

print(person)
# Output: {'name': 'Alice', 'age': 26, 'city': 'Dhaka'}
```

```
{'name': 'Alice', 'age': 26, 'city': 'Dhaka'}
```

In [126…
```
# update() – Update with another dictionary
student = {"name": "John", "age": 20}
new_info = {"course": "Python", "age": 21}
student.update(new_info)
print(student)  # Output: {'name': 'John', 'age': 21, 'course': 'Python'}
```

```
{'name': 'John', 'age': 21, 'course': 'Python'}
```

In [11]:
```
#If key exists: returns its value.
#If not: adds key with the given default value and returns it.
student = {'name': 'Taimur'}
```

```
# 'grade' doesn't exist, so it's added with default value 'A'
student.setdefault('grade', 'A')  # returns 'A'

# 'name' already exists, so it returns its value
student.setdefault('name', 'New Name')  # returns 'Taimur'

print(student)
# Output: {'name': 'Taimur', 'grade': 'A'}
```

```
{'name': 'Taimur', 'grade': 'A'}
```

Removing Elements

In [117…
```
# clear() - Remove all items
my_dict = {"name": "John", "age": 25}
my_dict.clear()
print(my_dict)  # Output: {}
```

```
{}
```

In [124…
```
# pop() - Remove an item
# Removes specified key and returns the value
student = {"name": "John", "age": 20}
removed_value = student.pop("age")
print(removed_value)  # Output: 20
print(student)  # Output: {'name': 'John'}
```

```
20
{'name': 'John'}
```

In [9]:
```
# popitem() - Remove an arbitrary item
# Removes and returns the last inserted key-value pair (in Python 3.7+)
student = {"name": "John", "age": 20}
removed_item = student.popitem()
print(removed_item)  # Output: ('age', 20)
print(student)  # Output: {'name': 'John'}
```

```
('age', 20)
{'name': 'John'}
```

Other Utilities

In [12]:
```
# Suppose you're designing a system where every user must have some default fiel
# but they haven't filled in anything yet.
# You can create a base template like this:

# Define the required profile fields
fields = ['name', 'email', 'phone', 'address']

# Set a default placeholder for all fields
user_profile = dict.fromkeys(fields, 'Not Provided')

print(user_profile)


# Set a default placeholder for all fields
```

```
user_profile = dict.fromkeys(fields, )

print(user_profile)
```

```
{'name': 'Not Provided', 'email': 'Not Provided', 'phone': 'Not Provided', 'addre
ss': 'Not Provided'}
{'name': None, 'email': None, 'phone': None, 'address': None}
```

Nested Dictionary

In [127…
```
# Nested Dictionaries
students = {
    "John": {"age": 20, "course": "Python"},
    "Alice": {"age": 22, "course": "Java"}
}
print(students["John"])  # Output: {'age': 20, 'course': 'Python'}
```

```
{'age': 20, 'course': 'Python'}
```

Functions

1. Functions are reusable blocks of code that perform a specific task.
2. Instead of writing the same code again and again, you write it once as a function and call it when needed.
3. Break down complex problems
4. Increase readability

User-defined Functions

In [13]:
```
def greet():
    print("Hello!")
```

In [ ]:
```
# def greet():

# def stands for define — it's how we create a function in Python.

# greet is the name of the function.

# () means this function takes no parameters (no input values for now).

# The : starts the function body (what the function will do).
```

In [15]:
```
# print("Hello!")
# This is the body of the function.

# When you call the function, it will print the message "Hello!" to the screen.
```

In [14]:
```
greet()
```

```
Hello!
```

Parameters and Arguments

In [16]:
```
def greet(name):          # 'name' is a parameter
    print("Hello", name)

greet('Taimur')           # 'Taimur' is an argument
```

```
Hello Taimur
```

In [17]:
```
## Keyword & Default Arguments
def greet(name, message):
    print(name, message)

greet(message="Hi", name="Taimur")
```

```
Taimur Hi
```

Function with Default Arguments

In [18]:
```
# Default Argument
def greet(name, message="Hello"):
    print(name, message)

greet("Taimur")           # Output: Taimur Hello
```

```
Taimur Hello
```

Return Statement

In [19]:
```
def add(x, y):
    return x + y

result = add(3, 4)
print(result)  # 7
```

```
7
```

Difference between print() and return

1. print() shows the result to the user
2. return sends the result back to the caller

Python Built-in functions

Type Conversion Functions:

1. int() – Converts a value to an integer.
2. float() – Converts a value to a float.
3. str() – Converts a value to a string.
4. bool() – Converts a value to a boolean (True/False).
5. list() – Converts an iterable (like a string, tuple, etc.) to a list.

In [21]:
```
x = "10"
y = int(x)  # Converts the string "10" to an integer 10
```

Mathematical Functions:

1. abs() – Returns the absolute value.
2. round() – Rounds a number to a specified number of decimals.
3. min() – Returns the smallest item in an iterable.
4. max() – Returns the largest item in an iterable.
5. sum() – Adds up all items in an iterable.

In [22]:
```
numbers = [2, 5, 8]
print(sum(numbers))  # Output: 15
```

```
15
```

Collection Functions:

1. len() – Returns the length (number of items) of an object.
2. sorted() – Returns a sorted list from an iterable.
3. reversed() – Returns an iterator that accesses the given iterable in reverse.
4. all() – Returns True if all elements are true.
5. any() – Returns True if any element is true.

In [23]:
```
numbers = [1, 2, 3]
print(len(numbers))  # Output: 3
```

```
3
```

Input/Output Functions:

1. input() – Reads a line from input and returns it as a string.
2. print() – Outputs the specified message to the console.

In [24]:
```
name = input("Enter your name: ")
print("Hello, " + name)
```

```
Enter your name: Taimur
Hello, Taimur
```

Other Utility Functions:

1. id() – Returns the unique id of an object.
2. type() – Returns the type of an object.
3. isinstance() – Checks if an object is an instance of a particular class or subclass.

In [25]:
```
x = 5
print(type(x))  # Output: <class 'int'>
```

```
<class 'int'>
```

Task: Create a function that takes a list of numbers and returns a dictionary with statistics: min, max, sum, and average.

In [20]:
```python
def calculate_stats(numbers):
    # Ensure the list is not empty
    if len(numbers) == 0:
        return "List is empty, cannot calculate stats."

    # Calculate stats
    stats = {
        'min': min(numbers),
        'max': max(numbers),
        'sum': sum(numbers),
        'average': sum(numbers) / len(numbers)  # Calculate the average
    }

    return stats

# Example usage
numbers_list = [5, 10, 15, 20, 25]
result = calculate_stats(numbers_list)

print(result)
```

```
{'min': 5, 'max': 25, 'sum': 75, 'average': 15.0}
```

Explanation:

The function calculate_stats() takes a list of numbers as input.

1. min(): Finds the smallest number in the list.
2. max(): Finds the largest number.
3. sum(): Adds up all the numbers.
4. average: Sum divided by the number of elements in the list.

If the list is empty, it returns a message indicating that stats can't be calculated.

Function with Variable-Length Arguments

Python allows you to define functions that accept an arbitrary number of arguments using *args (non-keyword arguments) or **kwargs (keyword arguments).

1. *args -> Accepts variable number of values (tuple)
2. **kwargs -> Accepts variable number of named arguments (dictionary)

In [15]:
```python
def add(*args):
    return sum(args)

print(add(1, 2, 3))  # Output: 6
```

In [16]:
```
def describe_pet(**kwargs):
    print(kwargs)

describe_pet(name="Bobby", type="dog")  # Output: {'name': 'Bobby', 'type': 'dog
```

```
{'name': 'Bobby', 'type': 'dog'}
```

Lambda Function

In [6]:
```
# It is used when you need a simple one-line function for a short period of time
lambda arguments: expression
```

Out[6]:
```
<function __main__.<lambda>(arguments)>
```

In [3]:
```
add = lambda x,y:x+y
```

In [5]:
```
add(5,3)
```

Out[5]:
```
8
```

In [7]:
```
def add(x, y):
    return x + y
```

In [8]:
```
add(5,3)
```

Out[8]:
```
8
```

Higher-Order Functions

Lambda functions are often used with:

1. map()
2. filter()
3. sorted()

In [11]:
```
numbers = [1,2,3,4]
squares = list(map(lambda x : x**2, numbers))
```

In [12]:
```
squares
```

Out[12]:
```
[1, 4, 9, 16]
```

In [13]:
```
# Keep only even numbers:

numbers = [1, 2, 3, 4, 5, 6]
evens = list(filter(lambda x: x % 2 == 0, numbers))
print(evens)  # Output: [2, 4, 6]
```

```
[2, 4, 6]
```

In [14]:
```
#Sort a list of tuples by the second value:
pairs = [(1, 3), (2, 2), (4, 1)]
```

```
sorted_pairs = sorted(pairs, key=lambda x: x[1])
print(sorted_pairs)  # Output: [(4, 1), (2, 2), (1, 3)]
```

```
[(4, 1), (2, 2), (1, 3)]
```

Nested Functions

In [17]:
```
def outer():
    def inner():
        print("This is the inner function")
    inner()

outer()  # Output: This is the inner function
```

```
This is the inner function
```

In [ ]:
```

```