10 Built-In Data Structures

DSA (**D**ata **S**tructures and **A**lgorithms) is the study of organizing data efficiently using data structures like arrays, stacks, and trees, paired with step-by-step procedures (or algorithms) to solve problems effectively.

There are two types of data structures

- In-built data structures
- User-defined data structures

Built-in Data Structures

Built-in data structures are the data strctures that provoded in the programming language.

- Lists
- Tuples
- Sets
- Dictionaries

Lists

Lists are ordered collection of data just like arrays in other programming languages. It allows different types of elements in the list, this is possible because a list mainly stores references at contiguous locations and actual items maybe stored at different locations.

- List can contain duplicate items.
- List in Python are Mutable. Hence, we can modify, replace or delete the items.
- List are ordered. It maintain the order of elements based on how they are added.
- Accessing items in List can be done directly using their position (index), starting from 0.

```
a = [10, 20, 15]

print(a[0]) # access first item
a.append(11) # add item
a.remove(20) # remove item

print(a)

# Output
```

```
10
[10, 15, 11]
```

Creating a List

Here are some common methods to create a list

Using Square Brackets

```
# List of integers
a = [1, 2, 3, 4, 5]

# List of strings
b = ['apple', 'banana', 'cherry']

# Mixed data types
c = [1, 'hello', 3.14, True]

print(a)
print(b)
print(c)

# Output
[1, 2, 3, 4, 5]
['apple', 'banana', 'cherry']
[1, 'hello', 3.14, True]
```

Using list() Constructor

We can also create a list by passing an **iterable**

```
# From a tuple
a = list((1, 2, 3, 'apple', 4.5))

print(a)

# Output
[1, 2, 3, 'apple', 4.5]
```

Creating List with Repeated Elements

We can create a list with repeated elements using the multiplication operator.

```
# Create a list [2, 2, 2, 2, 2]
a = [2] * 5

# Create a list [0, 0, 0, 0, 0, 0, 0]
b = [0] * 7

print(a)
print(b)

# Output
[2, 2, 2, 2, 2]
[0, 0, 0, 0, 0, 0, 0]
```

Accessing List Items

Elements in a list can be accessed using **indexing** Python indexes start at **0**, so **a[0]** will access the first element, while **negative indexing** allows us to access elements from the end of the list. Like index -1 represents the last elements of list

```
a = [10, 20, 30, 40, 50]

# Access first element
print(a[0])

# Access last element
print(a[-1])

# Output
10
50
```

Adding Elements into List

We can add elements to a list using the following methods

- **append()** - Adds an element at the end of the list
- **extend()** - Adds multiple elements to the end of the list
- **insert()** - Adds an element at a specific position

```
# Initialize an empty list
a = []

# Adding 10 to end of list
```

```
a.append(10)
print("After append(10):", a)

# Inserting 5 at index 0
a.insert(0, 5)
print("After insert(0, 5):", a)

# Adding multiple elements  [15, 20, 25] at the end
a.extend([15, 20, 25])
print("After extend([15, 20, 25]):", a)

# Output
After append(10): [10]
After insert(0, 5): [5, 10]
After extend([15, 20, 25]): [5, 10, 15, 20, 25]
```

Updating Elements into List

We can change the value of an element by accessing it using its index.

```
a = [10, 20, 30, 40, 50]

# Change the second element
a[1] = 25

print(a)

# Output
[10, 25, 30, 40, 50]
```

Removing Elements from List

We can remove elements from a list using

- `remove()` : Removes the first occurrence of an element.
- `pop()` : Removes the element at a specific index or the last element if no index is specified.
- `del` statement: Deletes an element at a specified index.

```
a = [10, 20, 30, 40, 50]

# Removes the first occurrence of 30
a.remove(30)
print("After remove(30):", a)
```

```
# Removes the element at index 1 (20)
popped_val = a.pop(1)
print("Popped element:", popped_val)
print("After pop(1):", a)

# Deletes the first element (10)
del a[0]
print("After del a[0]:", a)

# Output
After remove(30): [10, 20, 40, 50]
Popped element: 20
After pop(1): [10, 40, 50]
After del a[0]: [40, 50]
```

Iterating Over Lists

We can iterate the Lists easily by using iteration methods like `for` and `while` loops.

Using for Loop

```
a = ['apple', 'banana', 'cherry']

# Iterating over the list
for item in a:
    print(item)

# Output
apple
banana
cherry
```

Using while Loop

```
a = ['apple', 'banana', 'cherry']

i = 0
while i < len(a):
        print(a[i])
        i += 1

# Output
apple
```

```
banana
cherry
```

Nested Lists

A nested list is a list within another list, which is useful for representing matrices or tables.

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

# Access element at row 2, column 3
print(matrix[1][2])

# Output
6
```

List Methods

`append()` - Adds an element to the end of the list.

```
a = [1, 2, 3]

# Add 4 to the end of the list
a.append(4)
print(a)

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

`copy()` - Returns a shallow copy of the list.

```
a = [1, 2, 3]

# Create a copy of the list
b = a.copy()
print(b)

# Output
[1, 2, 3]
```

`clear()` - Removes all elements from the list.

```
a = [1, 2, 3]

# Remove all elements from the list
a.clear()
print(a)

# Output
[]
```

`count()` - Returns the number of times a specified element appears in the list.

```
a = [1, 2, 3, 2]

# Count occurrences of 2 in the list
print(a.count(2))

# Output
2
```

`extend` - Adds elements from another list to the end of the current list.

```
a = [1, 2]

# Extend list a by adding elements from list [3, 4]
a.extend([3, 4])
print(a)

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

`index()` - Returns the index of the first occurrence of a specified element.

```
a = [1, 2, 3]

# Find the index of 2 in the list
print(a.index(2))

# Output
1
```

`insert()` - Inserts an element at a specified position.

```python
a = [1, 3]

# Insert 2 at index 1
a.insert(1, 2)
print(a)

# Output
[1, 2, 3]
```

`pop()` - Removes and returns the element at the specified position (or the last element if no index is specified).

```python
a = [1, 2, 3]

# Remove and return the last element in the list
a.pop()
print(a)

# Output
[1, 2]
```

`remove()` - Removes the first occurrence of a specified element.

```python
a = [1, 2, 3]

# Remove the first occurrence of 2
a.remove(2)
print(a)

# Output
[1, 3]
```

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

```python
a = [1, 2, 3]

# Reverse the list order
a.reverse()
print(a)

# Output
[3, 2, 1]
```

`sort()` - Sorts the list in ascending order (by default).

```
a = [3, 1, 2]

# Sort the list in ascending order
a.sort()
print(a)

# Output
[1, 2, 3]
```

List Comprehension

List comprehension is a way to create lists using a concise syntax. It allows us to generate a new list by applying an **expression** to each item in an existing **iterable** (such as a **list** or **range**).

For example, if we have a list of integers and want to create a new list containing the square of each element.

```
a = [2,3,4,5]

res = [val ** 2 for val in a]

print(res)

# Output
[4, 9, 16, 25]
```

Syntax of list comprehension

`[` **expression** for **item** in **iterable** if **condition** `]`

- **expression**: The transformation or value to be included in the new list.
- **item**: The current element taken from the iterable.
- **iterable**: A sequence or collection (e.g., list, tuple, set).
- **if condition (optional)**: A filtering condition that decides whether the current item should be included.

for loop vs. list comprehension

Example: Let's take an example, where we want to double each number of given list into a new list.

Using a for loop

```
a = [1, 2, 3, 4, 5]

# Create an empty list 'res' to store results
res = []

# Iterate over each element in list 'a'
for val in a:

    # Multiply each element by 2 and append it to 'res'
    res.append(val * 2)

print(res)

# Output
[2, 4, 6, 8, 10]
```

Using list comprehension

```
a = [1, 2, 3, 4, 5]

res = [val * 2 for val in a]

print(res)

# Output
[2, 4, 6, 8, 10]
```

Conditional statements in list comprehension

List comprehensions can include conditional statements to filter or modify items based on specific criteria.

Example: Suppose we want to filter all even list from the given list.

```
a = [1, 2, 3, 4, 5]

res = [val for val in a if val % 2 == 0]

print(res)

# Output
[2, 4]
```

Examples of list comprehension

Creating a list from a range

```
# Creates a list of numbers from 0 to 9
a = [i for i in range(10)]

print(a)

# Output
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Using nested loops

List comprehension can also be used with nested loops.

```
# Creates a list of tuples representing all combinations of (x, y)
# where both x and y range from 0 to 2.
coordinates = [(x, y) for x in range(3) for y in range(3)]

print(coordinates)

# Output
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
```

Flattening a list of lists

Suppose we have a list of lists and we want to convert it into a single list.

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

res = [val for row in mat for val in row]

print(res)

# Output
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List Slicing

Python **list slicing** is fundamental concept that let us easily access specific elements in a list.

Example: Get the items from a list starting at position 1 and ending at position 4 (exclusive).

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

# Get elements from index 1 to 4 (excluded)
print(a[1:4])

# Output
[2, 3, 4]
```

List Slicing Syntax

list_name `[` start : end : step `]`

- **start (optional)** - Index to begin the slice (inclusive). Defaults to 0 if omitted.
- **end (optional)** - Index to end the slice (exclusive). Defaults to the length of list if omitted.
- **step (optional)** - Step size, specifying the interval between elements. Defaults to 1 if omitted.

List Slicing Examples

Get all the items from a list

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

# Get all elements in the list
print(a[::])
print(a[:])

# Output
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Get all items before/after a specific position

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

# Get elements starting from index 2
# to the end of the list
b = a[2:]
print(b)

# Get elements starting from index 0
# to index 3 (excluding 3th index)
c = a[:3]
```

```
print(c)

# Output
[3, 4, 5, 6, 7, 8, 9]
[1, 2, 3]
```

Get all items between two positions

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

# Get elements from index 1
# to index 4 (excluding index 4)
b = a[1:4]
print(b)

# Output
[2, 3, 4]
```

Get items at specified intervals

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

# Get every second element from the list
# starting from the beginning
b = a[::2]
print(b)

# Get every third element from the list
# starting from index 2 to 8(exclusive)
c = a[1:8:3]
print(c)

# Output
[1, 3, 5, 7, 9]
[2, 5, 8]
```

Out-of-bound slicing

list slicing allows out-of-bound indexing without raising errors. If we specify indices beyond the list length then it will simply return the available items.

Negative Indexing

Negative indexing is useful for accessing elements from the end of the list.

Extract elements using negative indices

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

# Get elements starting from index -2
# to end of list
b = a[-2:]
print(b)

# Get elements starting from index 0
# to index -3 (excluding 3th last index)
c = a[:-3]
print(c)

# Get elements from index -4
# to -1 (excluding index -1)
d = a[-4:-1]
print(d)

# Get every 2nd elements from index -8
# to -1 (excluding index -1)
e = a[-8:-1:2]
print(e)

# Output
[8, 9]
[1, 2, 3, 4, 5, 6]
[6, 7, 8]
[2, 4, 6, 8]
```

Reverse a list using slicing

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

# Get the entire list using negative step
b = a[::-1]
print(b)

# Output
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Tuples

A tuple in Python is an immutable ordered collection of elements. Tuples are similar to lists, but unlike lists, they cannot be changed after their creation (i.e., they are immutable).

Creating a Tuple

A tuple is created by placing all the items inside parentheses (), separated by commas.

```
# Creating an empty Tuple
tup = ()
print(tup)

# Using String
tup = ('Python', 'For')
print(tup)

# Using List
li = [1, 2, 4, 5, 6]
print(tuple(li))

# Using Built-in Function
tup = tuple('Python')
print(tup)

# Output
()
('Python', 'For')
(1, 2, 4, 5, 6)
('P', 'y', 't', 'h', 'o', 'n')
```

Creating a Tuple with Mixed Datatypes

```
# Creating a Tuple with Mixed Datatype
tup = (5, 'Welcome', 7, 'Python')
print(tup)

# Creating a Tuple with nested tuples
tup1 = (0, 1, 2, 3)
tup2 = ('python', 'code')
tup3 = (tup1, tup2)
print(tup3)

# Creating a Tuple with repetition
tup1 = ('Python',) * 3
print(tup1)
```

```
# Creating a Tuple with the use of loop
tup = ('Python')
n = 5
for i in range(int(n)):
    tup = (tup,)
    print(tup)

# Output
(5, 'Welcome', 7, 'Python')
((0, 1, 2, 3), ('python', 'code'))
('Python', 'Python', 'Python')
('Python',)
(('Python',),)
((('Python',),),)
(((('Python',),),),)
((((('Python',),),),),)
```

Python Tuple Operations

Below are the Python tuple operations.

- Accessing of Python Tuples
- Concatenation of Tuples
- Slicing of Tuple
- Deleting a Tuple

Accessing of Tuples

We can access the elements of a tuple by using indexing and slicing, similar to how we access elements in a list.

```
# Accessing Tuple with Indexing
tup = tuple("Python")
print(tup[0])

# Accessing a range of elements using slicing
print(tup[1:4])
print(tup[:3])

# Tuple unpacking
tup = ("Python", "For", "Students")

# This line unpack values of Tuple1
a, b, c = tup
```

```
print(a)
print(b)
print(c)

# Output
P
('y', 't', 'h')
('P', 'y', 't')
Python
For
Students
```

Concatenation of Tuples

Tuples can be concatenated using the + operator.

```
tup1 = (0, 1, 2, 3)
tup2 = ('Python', 'For', 'Students')

tup3 = tup1 + tup2
print(tup3)

# Output
(0, 1, 2, 3, 'Python', 'For', 'Students')
```

Note: Only the same datatypes can be combined with concatenation. an error arises if a list and a tuple are combined.

Slicing of Tuple

Slicing a tuple means creating a new tuple from a subset of elements of the original tuple. Negative Increment values can also be used to reverse the sequence of Tuples.

```
# Slicing of a Tuple with Numbers
tup = tuple('PYTHONFORSTUDENTS')

# Removing First element
print(tup[1:])

# Reversing the Tuple
print(tup[::-1])

# Printing elements of a Range
print(tup[4:9])
```

```
# Output
('Y', 'T', 'H', 'O', 'N','F', 'O', 'R', 'S', 'T', 'U', 'D', 'E', 'N', 'T',
'S')
('T', 'N', 'E', 'D', 'U', 'T', 'S', 'R', 'O', 'F', 'N', 'O', 'H', 'T', 'Y',
'P')
('O', 'N', 'F', 'O', 'R', 'S')
```

Deleting a Tuple

Since tuples are immutable, we cannot delete individual elements of a tuple. However, we can delete an entire tuple using `del` statement.

```
# Deleting a Tuple

tup = (0, 1, 2, 3, 4)
del tup

print(tup) # Gives error
```

Tuple Built-In methods

Tuples support only a few methods due to their immutable nature.

- `index()` - Find in the tuple and returns the index of the given value where it’s available
- `count()` - Returns the frequency of occurrence of a specified value

Tuple Built-In Functions

`all()` - Returns true if all element are true or if tuple is empty
`any()` - return true if any element of the tuple is true. if tuple is empty, return false
`len()` - Returns length of the tuple or size of the tuple
`enumerate()` - Returns enumerate object of tuple
`max()` - return maximum element of given tuple
`min()` - return minimum element of given tuple
`sum()` - Sums up the numbers in the tuple
`sorted()` - input elements in the tuple and return a new sorted list
`tuple()` - Convert an iterable to a tuple.

Sets

Python set is an unordered collection of multiple items having different datatypes. In Python, sets are mutable, unindexed and do not contain duplicates.

Creating a Set

In Python, the most basic and efficient method for creating a set is using curly braces.

```
set1 = {1, 2, 3, 4}
print(set1)

# Output
{1, 2, 3, 4}
```

Using the set() function

Python Sets can be created by using the built-in **set()** function with an iterable object or a sequence by placing the sequence inside curly braces, separated by a ‘comma’.

```
# Creating a Set
set1 = set()
print(set1)

set1 = set("PythonForStudents")
print(set1)

# Creating a Set with the use of a List
set1 = set(["Python", "for", "Students"])
print(set1)

# Creating a Set with the use of a tuple
tup = ("Python", "for", "Students")
print(set(tup))

# Creating a Set with the use of a dictionary
d = {"Python": 1, "for": 2, "Students": 3}
print(set(d))

# Output
set()
{'F', 'o', 'n', 'S', 'e', 'h', 'P', 't', 'y', 'd', 's', 'u', 'r'}
{'for', 'Students'}
{'for', 'Students'}
{'for', 'Students'}
```

Unordered, Unindexed and Mutability

In set, the order of elements is not guaranteed to be the same as the order in which they were added. The output could vary each time we run the program. Also the duplicate items entered are removed by itself.

Sets do not support indexing. Trying to access an element by index (set`[`0`]`) raises a `TypeError`.

```
# Creating a set
set1 = {3, 1, 4, 1, 5, 9, 2}

# Unordered: The order of elements is not preserved
print(set1)  # Output may vary: {1, 2, 3, 4, 5, 9}

# Unindexed: Accessing elements by index is not possible
# This will raise a TypeError
try:
    print(set1[0])
except TypeError as e:
    print(e)

# Output
{1, 2, 3, 4, 5, 9}
'set' object is not subscriptable
```

Adding Elements to a Set

We can add items to a set using `add()` method and `update()` method. `add()` method can be used to add only a single item. To add multiple items we use update() method.

```
# Creating a set
set1 = {1, 2, 3}

# Add one item
set1.add(4)

# Add multiple items
set1.update([5, 6])

print(set1)

# Output
{1, 2, 3, 4, 5, 6}
```

Accessing a Set

We can loop through a set to access set items as set is unindexed and do not support accessing elements by indexing.

```
set1 = set(["Python", "For", "Students"])

# Accessing element using For loop
for i in set1:
    print(i, end=" ")

# Checking the element# using in keyword
print("Python" in set1)

# Output
Python For Students True
```

Removing Elements from the Set

We can remove an element from a set in Python using several methods: remove(), discard() and pop().

Using remove() Method or discard() Method

remove() method removes a specified element from the set. If the element is not present in the set, it raises a KeyError. discard() method also removes a specified element from the set. Unlike remove(), if the element is not found, it does not raise an error.

```
# Using Remove Method
set1 = {1, 2, 3, 4, 5}
set1.remove(3)
print(set1)

# Attempting to remove an element that does not exist
try:
    set1.remove(10)
except KeyError as e:
    print("Error:", e)

# Using discard() Method
set1.discard(4)
print(set1)

# Attempting to discard an element that does not exist
set1.discard(10)  # No error raised
print(set1)
```

```
# Output
{1, 2, 4, 5}
Error: 10
{1, 2, 5}
{1, 2, 5}
```

Using pop() Method

pop() method removes and returns an arbitrary element from the set. This means we don't know which element will be removed. If the set is empty, it raises a KeyError.

```
set1 = {1, 2, 3, 4, 5}
val = set1.pop()
print(val)
print(set1)

# Using pop on an empty set
set1.clear()  # Clear the set to make it empty
try:
    set1.pop()
except KeyError as e:
    print("Error:", e)

# Output
1
{2, 3, 4, 5}
Error: 'pop from an empty set'
```

Using clear() Method

clear() method removes all elements from the set, leaving it empty.

```
set1 = {1, 2, 3, 4, 5}
set1.clear()
print(set1)

# Output
set()
```

Frozen Sets

A frozenset in Python is a built-in data type that is similar to a set but with one key difference that is immutability.

```
# Creating a frozenset from a list
fset = frozenset([1, 2, 3, 4, 5])
print(fset)

# Creating a frozenset from a set
set1 = {3, 1, 4, 1, 5}
fset = frozenset(set1)
print(fset)

# Output
frozenset({1, 2, 3, 4, 5})
frozenset({1, 3, 4, 5})
```

Dictionaries

A Python **dictionary** is a data structure that stores the value in **key**: **value** pairs. Values in a dictionary can be of any data type and can be duplicated, whereas keys can't be repeated and must be immutable.

```
d = {1: 'Python', 2: 'For', 3: 'Students'}
print(d)

# Output
{1: 'Python', 2: 'For', 3: 'Students'}
```

How to Create a Dictionary

A dictionary can be created by placing a sequence of elements within curly ****{}**** braces, separated by a 'comma'.

```
# create dictionary using { }
d1 = {1: 'Python', 2: 'For', 3: 'Students'}
print(d1)

# create dictionary using dict() constructor
d2 = dict(a = "Python", b = "for", c = "Students")
print(d2)

# Output
{1: 'Python', 2: 'For', 3: 'Students'}
{'a': 'Python', 'b': 'for', 'c': 'Students'}
```

Note: From Python 3.7 Version onward, Python dictionary are Ordered.

Accessing Dictionary Items

We can access a value from a dictionary by using the ***key*** within square brackets or `get()` method.

```
d = { "name": "Alice", 1: "Python", (1, 2): [1,2,4] }

# Access using key
print(d["name"])

# Access using get()
print(d.get("name"))

# Output
Alice
Alice
```

Adding and Updating Dictionary Items

We can add new key-value pairs or update existing keys by using assignment.

```
d = {1: 'Python', 2: 'For', 3: 'Students'}

# Adding a new key-value pair
d["age"] = 22

# Updating an existing value
d[1] = "Python dict"

print(d)

# Output
{1: 'Python dict', 2: 'For', 3: 'Students', 'age': 22}
```

Removing Dictionary Items

We can remove items from dictionary using the following methods.

- `del` : Removes an item by key.
- `pop()` : Removes an item by key and returns its value.
- `clear()` : Empties the dictionary.
- `popitem()` : Removes and returns the last key-value pair.

```
d = {1: 'Python', 2: 'For', 3: 'Students', 'age':22}

# Using del to remove an item
del d["age"]
print(d)

# Using pop() to remove an item and return the value
val = d.pop(1)
print(val)

# Using popitem to removes and returns
# the last key-value pair.
key, val = d.popitem()
print(f"Key: {key}, Value: {val}")

# Clear all items from the dictionary
d.clear()
print(d)

# Output
{1: 'Python', 2: 'For', 3: 'Students'}
Python
Key: 3, Value: Student
{}
```

Iterating Through a Dictionary

We can iterate over **keys** using `keys()` method, **values** using `values()` method or both using `item()` method with a `for` loop.

```
d = {1: 'Python', 2: 'For', 'age':22}

# Iterate over keys
for key in d:
    print(key)

# Iterate over values
for value in d.values():
    print(value)

# Iterate over key-value pairs
for key, value in d.items():
    print(f"{key}: {value}")

# Output
```

```
1
2
age
Python
For
22
1: Python
2: For
age: 22
```

Nested Dictionaries

Dictionaries inside dictionaries called nested dictionaries

```python
d = {1: 'Python', 2: 'For',
        3: {'A': 'Welcome', 'B': 'To', 'C': 'Python'}}

print(d)

# Output
{1: 'Python', 2: 'For', 3: {'A': 'Welcome', 'B': 'To', 'C': 'Python'}}
```