Data Structures in Python

1. Lists

Lists are one of the most commonly used data structures in Python. A list is a mutable, ordered sequence of items. You can create a list using the `list` keyword or the square brackets `[]`.

```
# Create a list using the list keyword
list1 = list()
print(list1) # []

# Create a list using square brackets
list2 = []
print(list2) # []

# Create a list with initial values
list3 = [1, 2, 3]
print(list3) # [1, 2, 3]
```

Since Python is a dynamically typed language, you can create lists with different types of values.

```
# Create a list with different types of values
list4 = [1, "Python", True]
print(list4) # [1, 'Python', True]
```

You can also use type hints to specify the type of values in a list.

```
from typing import List, Union

# Create a list with type hints
list5: List[int] = [1, 2, 3]

# Create a list of strings
list6: List[str] = ["Python", "Java", "C++"]

# Create a list of mixed types
list7: List[Union[int, str]] = [1, "Python", 3, "Java"]
```

You can access items in a list using the index operator `[]`. The index of the first item is `0`, the second item is `1`, and so on. You can also use negative indices to access items from the end of the list. The index of the last item is `-1`, the second last item is `-2`, and so on.

```
# Access items in a list
list8 = ["Python", "Java", "C++"]
print(list8[0]) # Python

# Access items from the end of the list
print(list8[-1]) # C++
```

You can either add items to the end of a list using the `append` method or insert items at a specific index using the `insert` method or the index operator `[]`.

```
# Add items to the end of a list
list9 = ["Python", "Java", "C++"]
list9.append("JavaScript")

# Insert items at a specific index
list9.insert(0, "TypeScript")

# Insert items at a specific index using the index operator
list9[0] = "Go"

print(list9) # ['Go', 'Python', 'Java', 'C++', 'JavaScript']
```

You can even slice a list using the slice operator `[:]`. The slice operator returns a new list containing the items in the specified range.

```
# Slice a list
list10 = ["Python", "Java", "C++", "JavaScript"]
print(list10[1:3]) # ['Java', 'C++']
```

You can find more information about lists here.

Internal working of lists

A `list` in Python is a dynamic array that can resize itself during runtime to accommodate a varying number of elements. The key features of dynamic arrays in Python include dynamic resizing, indexing, and support for various operations.

Here's an overview of how dynamic arrays work in Python:

1. **Dynamic Resizing:**
    - When elements are added to a list, Python dynamically allocates memory to store those elements.
    - The list has an initial capacity, but as elements are appended, it may need to resize itself to accommodate more elements.
    - Python often uses a strategy of doubling the capacity when resizing to amortize the cost of resizing over a sequence of appends, ensuring efficient performance.

2. **Indexing:**
    - Lists in Python support constant-time indexing, allowing you to access elements by their position in the list.
3. **Appending Elements:**
    - Appending elements to a list is generally an amortized constant-time operation. However, occasional resizing may lead to a linear-time operation in some cases.
4. **Deleting Elements:**
    - Deleting elements from the end of the list using `pop()` is a constant-time operation.

```
my_list = [1, 2, 3, 4]
my_list.pop()
```

    - Deleting elements from arbitrary positions may require shifting subsequent elements, resulting in a linear-time operation in the worst case.

```
my_list = [1, 2, 3, 4]
del my_list[1]
```

5. **Slicing:**
    - Slicing a list to create a new list is a constant-time operation, as it creates a new reference to the existing elements.
6. **Concatenation:**
    - Concatenating two lists using the `+` operator creates a new list and takes time proportional to the sum of the lengths of the lists.

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
concatenated_list = list1 + list2
```

2. Set

A set is a mutable, unordered collection of unique items. You can create a set using the `set` keyword or the curly braces `{}`.

```
# Create a set using the set keyword
set1 = set()

# Create a set using curly braces
set2 = {}

# Create a set with initial values
set3 = {1, 2, 3}
```

You can add new items to a set using the `add` method.

```
# Add new items to a set
set4 = {1, 2, 3}

# Add a new item
set4.add(4)

print(set4) # {1, 2, 3, 4}
```

You can also use the `update` method to add multiple items to a set.

```
# Add multiple items to a set
set5 = {1, 2, 3}

# Add multiple items
set5.update([4, 5, 6])

print(set5) # {1, 2, 3, 4, 5, 6}
```

Sets do not allow duplicate items. If you try to add a duplicate item to a set, it'll be ignored.

```
# Sets do not allow duplicate items
set6 = {1, 2, 3}

# Add a duplicate item
set6.add(1)

print(set6) # {1, 2, 3}
```

Sets do not also support indexing. If you try to access an item using an index, you'll get a `TypeError`.

```
# Sets do not support indexing
set7 = {1, 2, 3}

# TypeError: 'set' object is not subscriptable
print(set7[0])
```

Sets allow membership testing using the `in` and `not in` operators.

```
# Membership testing
set8 = {1, 2, 3}

print(1 in set8) # True

print(4 not in set8) # True
```

They also support set operations like union, intersection, difference, and symmetric difference.

```
# Set operations
set9 = {1, 2, 3}
set10 = {3, 4, 5}

# Union
print(set9 | set10) # {1, 2, 3, 4, 5}

# Intersection
print(set9 & set10) # {3}

# Difference
print(set9 - set10) # {1, 2}

# Symmetric difference
print(set9 ^ set10) # {1, 2, 4, 5}
```

You can read more about sets here.

Internal working of sets

1. **Hash Table:**
    - A set in Python is implemented as a hash table, which is a data structure that allows for efficient insertion, deletion, and retrieval of elements.
    - The primary advantage of using a hash table is its ability to provide constant-time average complexity for common operations, such as adding, removing, and checking for membership.
2. **Hashing:**
    - Hashing is a process that converts an element (such as a string or a number) into a fixed-size numerical value, known as a hash code.
    - Python's `hash()` function is used to compute the hash code for elements.

    ```
    my_set = {1, 2, 3}
    hash_value = hash(2)
    ```

3. **Bucket Array:**

- Internally, a set maintains an array of "buckets" or "slots," where each bucket is capable of holding multiple elements.
- The size of the array and the number of buckets are determined dynamically based on the number of elements in the set.

4. **Insertion:**
    - When an element is added to the set, its hash code is computed.
    - The hash code is then used to determine the bucket in which the element should be placed.
    - If the bucket is empty, the element is added directly. If the bucket is not empty (collision), a collision resolution mechanism is employed.
5. **Collision Resolution:**
    - In the case of a collision (two elements hashing to the same bucket), Python uses a technique called open addressing, specifically quadratic probing.
    - Quadratic probing involves searching for the next available slot by using a quadratic function of the form `(h + i^2) % N`, where `h` is the original hash code, `i` is the attempt number, and `N` is the size of the array.
6. **Deletion:**
    - When an element is removed from the set, its hash code is computed, and the set searches for the corresponding bucket.
    - If the element is found, it is removed. If there are other elements in the same bucket, their positions may need to be adjusted.
7. **Retrieval:**
    - When checking for membership in a set, the hash code of the queried element is computed, and the set looks in the corresponding bucket.
    - If the element is found, it is present in the set; otherwise, it is not.

3. Dictionary

A dictionary is a mutable, unordered collection of key-value pairs. You can create a dictionary using the `dict` keyword or the curly braces `{}`.

```
# Create a dictionary using the dict keyword
dict1 = dict()

# Create a dictionary using curly braces
dict2 = {}

# Create a dictionary with initial values
dict3 = {"first_name": "Tantia", "last_name": "Tope"}
```

Similar to lists, the data types of keys and values in a dictionary can be different.

```
# Create a dictionary with different types of keys and values
dict4 = {"name": "Tantia Tope", "age": 50, "is_employed": True}
```

You can also use type hints to specify the types of keys and values in a dictionary.

```
from typing import Dict, Union

# Create a dictionary with type hints
dict5: Dict[str, Union[str, int]] = {"name": "Tantia Tope", "age": 50}
```

You can access values in a dictionary using the key. If the key doesn't exist, you'll get a `KeyError`.

```
# Access values in a dictionary
dict6 = {"name": "Tantia Tope", "age": 50}

print(dict6["name"]) # Tantia Tope

# KeyError: 'address'
print(dict6["address"])
```

You can also use the `get` method to access values in a dictionary. If the key doesn't exist, you'll get `None`.

```python
# Access values in a dictionary using the get method
dict7 = {"name": "Tantia Tope", "age": 50}

print(dict7.get("name")) # Tantia Tope
print(dict7.get("address")) # None
```

You can add new key-value pairs to a dictionary using the `[]` operator or the `update` method.

```python
# Add new key-value pairs to a dictionary
dict8 = {"name": "Tantia Tope", "age": 50}

# Add a new key-value pair using the [] operator
dict8["address"] = "Meerut"

# Add a new key-value pair using the update method
dict8.update({"is_employed": True})

print(dict8) # {'name': 'Tantia Tope', 'age': 50, 'address': 'Meerut', 'is_employed': True
```

You can read more about dictionaries here.

> **defaultdict**

````
The `defaultdict` class is a subclass of the `dict` class that provides a

```python
from collections import defaultdict

# Create a defaultdict
dict9 = defaultdict(int)

# Add a new key-value pair
dict9["age"] = 50

# Access a key that doesn't exist
print(dict9["address"]) # 0
```

In this example, the `defaultdict` returns the default value `0` for the
The return value is `0` because we passed `int` as the default value when

You can read more about it [here](https://docs.python.org/3/library/colle
````

Dictionaries are also implemented as hash tables in Python.

4. Tuple

A tuple is an immutable, ordered sequence of items. A tuple is different from a list in that it cannot be modified after creation. You can create a tuple using the `tuple` keyword or the parentheses `()`.

```python
# Create a tuple using the tuple keyword
tuple1 = tuple()

# Create a tuple using parentheses
tuple2 = ()

# Create a tuple with initial values
tuple3 = (1, 2, 3)
```

You can also use type hints to specify the type of values in a tuple.

```
from typing import Tuple, Union

# Create a tuple with type hints
tuple4: Tuple[int, str, bool] = (1, "Python", True)
```

You can access items in a tuple using the index operator `[]`. The index of the first item is `0`, the second item is `1`, and so on. You can also use negative indices to access items from the end of the tuple. The index of the last item is `-1`, the second last item is `-2`, and so on.

```
# Access items in a tuple
tuple5 = (1, 2, 3)

print(tuple5[0]) # 1
print(tuple5[-1]) # 3
```

You can destruct a tuple using the assignment operator `=`. This allows you to assign each item in a tuple to a separate variable.

```
# Destruct a tuple
tuple6 = (1, 2, 3)

# Assign each item in the tuple to a separate variable
a, b, c = tuple6

print(a, b, c) # 1 2 3
```

Tuples are extremely useful when you want to return multiple values from a function.

```
# Return multiple values from a function
def get_name() -> Tuple[str, str]:
    return "Tantia", "Tope"

first_name, last_name = get_name()
print(first_name, last_name) # Tantia Tope
```

You can read more about tuples here.

> **The collections module**
>
> The `collections` module provides a number of useful, advanced data structures that are not built-in to Python. The most commonly used data structures in this module apart from `defaultdict` are:
>
> - `Counter`: A dictionary that counts the number of occurrences of each item.
> - `OrderedDict`: A dictionary that remembers the order of insertion of items.
> - `deque`: A double-ended queue that supports adding and removing items from both ends.
>
> You can read more about the `collections` module here.

Comprension

Comprehension is a syntactic construct in Python that enables the concise creation of sequences (lists, dictionaries, and sets) using a single line of code. It combines the `for` loop and an optional `if` condition to generate elements dynamically based on specific criteria.

List Comprehension:

List comprehensions allow for the creation of lists in a concise and readable manner. The general syntax is as follows:

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

- `expression`: The expression to be evaluated and included in the list.
- `item`: The variable representing each element in the iterable.
- `iterable`: The source of elements for the list.
- `condition` (optional): An optional condition to filter elements.

Example:

```
squares = [x**2 for x in range(1, 6)]
# Output: [1, 4, 9, 16, 25]
```

Similarly, you can use a list comprehension to filter elements based on a condition.

```
even_squares = [x**2 for x in range(1, 6) if x % 2 == 0]
# Output: [4, 16]
```

You can also use nested loops in list comprehensions.

```
pairs = [(x, y) for x in range(1, 3) for y in range(1, 3)]

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

Dictionary Comprehension:

Dictionary comprehensions provide a concise way to create dictionaries. The syntax is similar to list comprehensions but uses key-value pairs enclosed in curly braces `{}`.

```
{key_expression: value_expression for item in iterable if condition}
```

- `key_expression`: The expression for the dictionary keys.
- `value_expression`: The expression for the corresponding values.
- `item`, `iterable`, and `condition` have the same meanings as in list comprehensions.

Example:

```
squares_dict = {x: x**2 for x in range(1, 6)}
# Output: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Set Comprehension:

Set comprehensions generate sets using a syntax similar to list comprehensions, but with

curly braces `{}`.

```
{expression for item in iterable if condition}
```

- `expression`: The expression to be evaluated and included in the set.
- `item`, `iterable`, and `condition` have the same meanings as in list comprehensions.

Example:

```
squares_set = {x**2 for x in range(1, 6)}
# Output: {1, 4, 9, 16, 25}
```

Advantages of Comprehensions:

1. **Conciseness:** Comprehensions offer a concise and readable syntax, reducing the need for explicit loops and temporary variables.
2. **Readability:** Comprehensions convey the intent of the code more clearly, making it easier to understand.
3. **Performance:** Comprehensions are often more efficient than equivalent loops, especially for simple operations.