# Working with Data Structures 2

# Summary

- Data Types Properties
- Tuples
- Sets
- Dictionaries

levi nine

# Data Types

# Data Types Properties

- Mutability
- Immutability
- Hashable

# Mutability

A mutable object can be changed after it is created — its content can be modified.

```
my_list = [1, 2, 3]
my_list[0] = 99      # Modifies the list in place
print(my_list)       # Output: [99, 2, 3]
```

Common Mutable Types: Lists, Dictionaries, Sets

Great when you need to store collections that will change dynamically.

# Immutability

An immutable object cannot be changed after it is created – any change results in a new object.

```
my_tuple = (1, 2, 3)
my_tuple[0] = 99                # This would raise a TypeError

name = "Alice"
name = name.replace("A", "M")
print(name)                     # "Mlice" (new string created)
```

Common Mutable Types: Integer, Float, Strings, Tuples

Used in concurrent programming and as **dictionary keys**

# Hashability

An object is hashable if it has a hash value that does not change during its lifetime.

Hashable = Immutable + __hash__( ) method

```
# Immutable and hashable
my_dict = {(1, 2): "tuple key"}

# Tuple can be a dict key
print(my_dict[(1, 2)])          # "tuple key"

# Unhashable example
my_dict = {[1, 2]: "list key"}  # Raises TypeError, list is unhashable
```

Common Hashable Types: Integer, Float, Strings, Tuples(if all elements of the tuple are also hashable)

Needed when using objects as keys in dictionaries or elements in sets.

levi nine

# Tuples

# Tuples

A **tuple** in Python is a collection that can store multiple items in a single variable. Think of it like a box that holds values—like numbers, words, or even other tuples—but once packed, you can't change what's inside.

- Tuples are ordered: Items have a fixed position (called an index).
- Tuples are unchangeable (immutable): Once created, you can’t add, change, or remove items.
- Tuples can contain different data types: numbers, strings, booleans, etc.
- Tuples are written with parentheses: ()

# Tuples

- Similar with lists but immutable

```
coords = (10, 20)
coords[0]         # 10
coords[0] = 30    # TypeError - tuple is immutable
```

# Tuples

- Tuples can be hybrid

```
my_tuple = tuple()                  # empty tuple
my_tuple = ()                       # empty tuple
my_tuple = (1, "hello", 3.14)       # tuple with values
my_tuple = (1, )                    # tuple containing (1)
my_tuple = (1, 2) * 3               # tuple containing (1,2, 1,2, 1,2)
```

- Unpacking

```
x, y = coords
print(x, y)   # 10 20
```

# Tuples

- Accessing elements can be done in the same way as for lists

```
my_tuple = (1, "hello", 3.14)

my_tuple[0]                    # 1
my_tuple[-1]                   # 3.14
my_tuple[-2]                   # 'hello'
my_tuple[:2]                   # (1, 'hello')
my_tuple[1:]                   # ('hello', 3.14)
my_tuple[0:2]                  # (1, 'hello')
my_tuple[1:-1]                 # ('hello')
```

# Tuples

- Tuples can be concatenated

```
my_tuple = (1, "hello", 3.14)

new_tuple = my_tuple + (5, "bye", True)
print(new_tuple) #(1, 'hello', 3.14, 5, 'bye', True)
```

- Can’t concatenate a tuple with a list!

# Tuples

- Tuples can be used to simulate a matrix

```
matrix = (
    (1, 2, 3),
    (4, 5, 6)
)

print(matrix[0][1])  # 2 (row 0, column 1)
print(matrix[1][2])  # 6 (row 1, column 2)

matrix[1][1] = 99    # TypeError
```

# Tuples

- Tuples are also used to return multiple values from a function.

```
def calculate_sum_and_product(numbers):
    total_sum = sum(numbers)
    total_product = 1
    for num in numbers:
        total_product *= num
    return (total_sum, total_product)

result = calculate_sum_and_product([2, 3, 4])
print(result)  # (9, 24)
```

# Tuples

- Tuples can be iterated with for keyword

```
for i in (10, 20, 30, 40, 50):
  print(i)                          # 10 20 30 40 50
```

- To find out how many items are in a tuple the built-in len() function can be used

```
nums = (10, 20, 30, 40, 50)
print(len(nums))                    # 5
```

# Sets

# Sets

A **set** in Python is a collection used to store multiple items in a single variable — like a bag that holds things, but it doesn't care about the order, and it doesn’t allow duplicates.

- Sets are unordered: Items don't have a fixed position or index, and the order may change every time you access it.
- Sets are mutable: You can add, remove, or update items after the set is created.
- Sets do not allow duplicates: If you add a value that’s already in the set, it will be ignored.
- Sets can contain different data types: numbers, strings, booleans, etc. (as long as they're hashable).
- Sets are written with curly braces: { }

# Sets

- Collection of unique elements

```
my_list = [1, 2, 2, 3, 3, 3]
unique = set(my_list)                  # {1, 2, 3}

my_set = set()
my_set = {1, 2, 3}
my_set = {1, 2, 2, 1, 1, 3}            # {1, 2, 3}
my_set = {1, 2, "aa", 1, "AA", 3}      # {1, 2, 3, "AA", "aa"}
my_set = set((1, 1, 3, 2))             # {1, 2, 3}
my_set = set("Hello")                  # {'h', 'e', 'l', 'o'}
```

# Sets

- Elements of a set can’t be accessed, and two sets doesn’t support the addition operation

```
my_set = {1, 2, 3}

my_set[0]          # TypeError
my_set[1]          # TypeError

my_set + {0, 10}   # TypeError
```

# Sets

- Sets offer several functions to modify their contents, including the add method for adding a single element, the update method for adding multiple elements, and the |= operator for the same purpose

```
my_set = {1, 2, 3}

my_set.add(4)                  # {1, 2, 3, 4}
my_set.update({5, 6})          # {1, 2, 3, 4, 5, 6}
my_set.update({5}, {7, 8})     # {1, 2, 3, 4, 5, 6, 7, 8}

my_set |= {10}                 # {1, 2, 3, 4, 5, 6, 7, 8, 10}
```

# Sets

- To remove an element from a set can be used remove or discard. Remove throws an error if the element it's not in the set
- Clear it's used to empty an entire set

```
my_set = {1, 2, 3}

my_set.remove(2)    # {1, 3}
my_set.remove(5)    # KeyError

my_set.discard(3)   # {1}
my_set.discard(3)   # {1}

my_set.clear()      # {}
```

# Sets

- Union operation can be performed by using the operator | or the method union

```
a = {1, 2, 3}
b = {3, 4, 5}

c = a | b                                  # {1, 2, 3, 4, 5}

d = c.union({10}, {10, 11, 12})    # {1, 2, 3, 4, 5, 10, 11, 12}
```

# Sets

- Intersection can be performed by using the operator & or method intersection

```
a = {1, 2, 3}
b = {3, 4, 5}
c = {5, 6, 7}

d = a & b                          # {3}

e = b.intersection(c)              # {5}
f = c.intersection(e, {4, 8})      # {}
```

# Sets

- For difference exists the operator - or also the method difference

```
a = {1, 2, 3}
b = {3, 4, 5}

c = a - b                          # {1, 2}

d = c.difference({2})              # {1}
e = c.difference({2}, {1})         # {}
```

# Sets

- Symmetric difference can be performed using ^ or symmetric_difference method

```
a = {1, 2, 3, 4}
b = {3, 4, 5}

c = a ^ b                                  # {1, 2, 5}

d = c.symmetric_difference(b)              # {1, 2, 3, 4}
e = c.symmetric_difference(b, {1})         # TypeError, just one argument
```

# Sets

- To check if an element exists in a set we can use operator in and not in. The length of a set it's found using len method

```
s = {1, 2, 3}

x = 2 in s        # True
y = 4 not in s    # True

l = len(s)        # 3
```

# Sets

- To check if a set has no common elements with another one exists method isdisjoint

```
a = {1, 2}
b = {3, 4}

c = a.isdisjoint(b)  # True
```

- If a check is included in another there are issubset and is supers set methods and also <= and >= operators

```
a = {1, 2}
b = {1, 2, 3}

a.issubset(b)    # True
a <= b

b.issuperset(a)  # True
b >= a
```

# Set Comprehension

- Just like for lists, we can create sets in one line using {} instead of []

```
a = {_ for _ in range(5)}                          # {0, 1, 2, 3, 4}
b = {number for number in range(5) if number % 2 == 0} # {0, 2, 4}
c = {i*i for i in range(1, 6)}                     # {1, 4, 9, 16, 25}
```

# Set with Built-in functions

- The default built-in functions like map, filter, sorted, min or others can be used also with sets.

```
x = set(filter(lambda i: i%2==0, [1, 2, 3, 4, 5]))            # {2, 4}
y = set(map(lambda element: element * element, {1, 2, 3, 4})) # {1, 4, 9, 16}
z = set(filter(lambda x: x%5 == 1, range(20)))                # {1, 6, 11, 16}
```

# FrozenSet

- A frozenset is an immutable version of a set. Once created, you cannot add or remove elements.
- Useful for using as dictionary key, storing sets in other sets, ensuring read-only access

```
fs = frozenset([1, 2, 3])


print(fs)       # frozenset({1, 2, 3})
print(2 in fs)  # True
fs.add(4)       # AttributeError: 'frozenset' object has no attribute 'add'
```

# Dictionaries

# Dictionaries

A **dictionary** in Python is a collection that stores data in **key-value** pairs, similar to a hashmap in java or a JSON object.

- Dictionaries are unordered (they maintain insertion order, but conceptually they're not position-based)
- Items are accessed by their keys, not by position (index)
- Dictionaries are changeable (mutable)
- Dictionaries can have different data types for keys and values: Keys are usually strings or numbers (must be immutable), and values can be anything
- Dictionaries are written with curly braces: {}

# Dictionaries

- Dictionaries can be created in multiple ways

```
my_dict = dict()                          # empty dictionary
my_dict = {}                              # empty dictionary
my_dict = {"key1": 1, "key2": 2}          # dictionary with 2 keys: key1 and key2

my_dict = dict(a=1, b=2)                  # {'a': 1, 'b': 2}
my_dict = dict({"a":1, "b":2})            # {'a': 1, 'b': 2}
my_dict = dict([("a", 1), ("b", 2)])      # {'a': 1, 'b': 2}
my_dict = dict((("a", 1), ("b", 2)))      # {'a': 1, 'b': 2}
my_dict = dict(zip(["a", "b"], [1, 2])) # {'a': 1, 'b': 2}
my_dict = dict.fromkeys(["a", "b"], 2)  # {'a': 2, 'b': 2}
```

# Dictionaries

- Accessing an element of a dictionary can be made with [] or method get

```
person = {"name": "Alice", "age": 30}

person["name"]                      # Alice

person.get("name")                  # Alice
person.get("country", "Not Found") # Not Found
person.get("address")               # None
```

# Dictionaries

- An element can be added directly with [] operator or using update and setdefault method

```
person = {"name": "Alice"}

person["city"] = "Iasi"     # {'name': 'Alice', 'city': 'Iasi'}

person.update({"age": 30})  # {'name': 'Alice', 'city': 'Iasi', 'age': 30}

person.setdefault("h", 164) # {'name': 'Alice', 'city': 'Iasi', 'age': 30, 'h': 164}
```

# Dictionaries

- To delete a value exists del operator or pop method. Also clear to empty the entire dictionary

```
person = {"name": "Alice", "age": 30, "city": "Iasi"}

del person["age"]
print(person)                # {'name': 'Alice', 'city': 'Iasi'}

name = person.pop("name")  # Alice
val = person.pop("key", 0) # 0
val = person.pop("key")    # KeyError, no default provided

person.clear()               # {}
```

# Dictionaries

- There are methods used to retrieve the keys and values of a dictionary

```
person = {"name": "Alice", "age": 30}

person.keys()                    # dict_keys(['name', 'age'])
person.values()                  # dict_values(['Alice', 30])

check = "name" in person.keys() # True
```

# Dictionaries

- A dictionary can be iterate in for loops using items

```
person = {"name": "Alice", "age": 30}

for key in person.keys():
print(key, person[key])
# name Alice
# age 30


for key, value in person.items():
print(key, value)
# name Alice
# age 30
```

# Dictionaries

- A dictionary can be iterate in for loops also with enumerate

```
person = {"name": "Alice", "age": 30}

for elements in enumerate(person):
print(elements)
# (0, 'name')
# (1, 'age')
```

# Dictionaries

- Dictionaries can be concatenated in multiple ways

```
dict1 = {"a": 1, "b": 2}
dict2 = {"b": 3, "c": 4}

result1 = {**dict1, **dict2} # {'a': 1, 'b': 3, 'c': 4}
result2 = dict1 | dict2 # {'a': 1, 'b': 3, 'c': 4}

dict1.update(dict2) # {'a': 1, 'b': 3, 'c': 4} modify dict1
```

# Dictionaries

- Python also supports nested dictionaries

```
students = {
      "Alice": {"math": 90, "science": 85},
      "Bob": {"math": 75, "science": 80}
      }

print(students["Alice"]["math"]) # 90
```

# Dictionary Comprehension

- Like lists and sets also for dictionary exists the concept of comprehension

```
numbers = [0, 1, 2, 3, 4]
squares = {}
for x in numbers:
    squares[x] = x**2


squares = {x : x**2 for x in range(5)}
# 0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

# Default dict

```
from collections import defaultdict

my_defaultdict = defaultdict()
my_defaultdict.update({"aloha": 1})

for i in range(0, 4):
    my_defaultdict["aloha"] += 1

print(my_defaultdict)         # defaultdict(None, {'aloha': 5})
```

# Dictionary of frequencies

- Method 1: using classic dictionary

```
freq = {}
words = ['Aaa', 'Aa', 'AAA', 'aa', 'AAA', 'Aa', 'Aa', 'B']

for w in words:
  if w in freq.keys():
     freq[w] = 1 + freq[w]
  else:
     freq[w] = 1

print(freq) # {'Aaa': 1, 'Aa': 3, 'AAA': 2, 'aa': 1, 'B': 1}
```

# Dictionary of frequencies

- Method 2: classic dictionary without verification

```
freq = {}
words = ['Aaa', 'Aa', 'AAA', 'aa', 'AAA', 'Aa', 'Aa', 'B']

for w in words:
  freq[w] = 1 + freq.get(w, 0)

print(freq)  # {'Aaa': 1, 'Aa': 3, 'AAA': 2, 'aa': 1, 'B': 1}
```

# Dictionary of frequencies

- Method 3: default dict

```
from collections import defaultdict

freq= defaultdict(int)
words = ['Aaa', 'Aa', 'AAA', 'aa', 'AAA', 'Aa', 'Aa', 'B']

for w in words:
  freq[w] += 1

print(freq)
# defaultdict(<class 'int'>, {'Aaa': 1, 'Aa': 3, 'AAA': 2, 'aa': 1, 'B': 1})
```

# Dictionary of frequencies

- Method 4: Counter

```
from collections import Counter

print(Counter(words)) # Counter({'Aa': 3, 'AAA': 2, 'Aaa': 1, 'aa': 1, 'B': 1})
```

# Dictionary of frequencies

- Method 5: dictionary comprehension

```
print({i:words.count(i) for i in set(words)})

# {'aa': 1, 'AAA': 2, 'Aa': 3, 'B': 1, 'Aaa': 1}
# {'AAA': 2, 'B': 1, 'Aa': 3, 'aa': 1, 'Aaa': 1}
# It's different from run to run because sets are unordered!!
```

# Dictionary of frequencies

- Method 6: dictionary comprehension but fixed

```
print({i:words.count(i) for i in dict.fromkeys(words)})

# {'Aaa': 1, 'Aa': 3, 'AAA': 2, 'aa': 1, 'B': 1}
# Keep the initial order
```