CH-6: Dictionaries and Sets

Introduction

- built-in sequence collections—strings, lists and tuples
- built-in non-sequence collections—dictionaries and sets
- A dictionary is an unordered collection which stores key–value pairs that map immutable keys to values, just as a conventional dictionary maps words to definitions.
- A set is an unordered collection of *unique* immutable elements.

Dictionaries

- A dictionary associates keys with values.
- Each key maps to a specific value.

Keys	Key type	Values	Value type
Country names	`str`	Internet country codes	`str`
Decimal numbers	`int`	Roman numerals	`str`
States	`str`	Agricultural products	list of `str`
Hospital patients	`str`	Vital signs	tuple of `int`s and `float`s
Baseball players	`str`	Batting averages	`float`
Metric measurements	`str`	Abbreviations	`str`
Inventory codes	`str`	Quantity in stock	`int`

- A dictionary's keys must be immutable (such as strings, numbers or tuples) and unique(that is, no duplicates).
- Multiple keys can have the same value, such as two different inventory codes that have the same quantity in stock.

Dictionaries

- Creating a dictionary:
  - You can create a dictionary by enclosing in curly braces, $\{\}$, a comma-separated list of key-value pairs, each of the form *key: value*.
  - You can create an empty dictionary with $\{\}$.
- Key features:
  - Keys are unique: No duplicate keys allowed.
  - Keys must be immutable: Examples include strings, numbers, or tuples.
  - Values can be of any data type: Strings, lists, other dictionaries, etc.

Dictionaries

- Creating a dictionary

```
# Creating a dictionary
student = {" name " : " John Doe " ,
    " age " : 21 ,
    " branch " : " CSE " ,
    " grades " : [85 , 90 , 92]
}
```

- Because dictionaries are unordered collections, the display order can differ from the order in which the key–value pairs were added to the dictionary.

Dictionaries

- Determining if a Dictionary Is Empty
  - The built-in function `len` returns the number of key–value pairs in a dictionary:

```
In [6]: len(student)
Out[6]: 4
```

  - You can use a dictionary as a condition to determine if it's empty-a non-empty dictionary evaluates to `True` and an empty dictionary evaluates to `False`.

```
In [8]: if student:
   ...:     print("student dictionary is non-empty.")
   ...: else:
   ...:     print("student dictionary is empty.")
   ...:
student dictionary is non-empty.
```

Dictionaries

- We call method clear to delete the dictionary's key–value pairs

```
In [10]: if student:
    ...:     print("student dictionary is empty.")
    ...: else:
    ...:     print("student dictionary is empty.")
    ...:
student dictionary is empty.
```

- Iterating through a Dictionary
  - Consider the following dictionary

```
In [11]: days_per_month = { 'Jan': 31, 'Feb': 28, 'Mar': 31}

In [12]: days_per_month.items()
Out[12]: dict_items([('Jan', 31), ('Feb', 28), ('Mar', 31)])

In [13]: for month, days in days_per_month.items():
    ...:     print(f"{month}-{days}")
    ...:
Jan-31
Feb-28
Mar-31
```

Dictionaries

- Basic Dictionary Operations
  - consider the following dictionary-

```
In [1]: dict_alph = {'a':1, 'b':2, 'c':3, 'd'
    :4, 'e':5}
```

  - Accessing the Value Associated with a Key

```
In [4]: dict_alph['c']
Out[4]: 3
```

  - Updating the Value of an existing key-value pair

```
In [6]: dict_alph
Out[6]: {'a': 1, 'b': 2, 'c': 30, 'd': 4, 'e':
     5}
```

Dictionaries

Adding a New Key–Value Pair

```
In [7]: dict_alph['f'] =6

In [8]: dict_alph
Out[8]: {'a': 1, 'b': 2, 'c': 30, 'd': 4, 'e': 5, 'f': 6}
```

Removing a Key–Value Pair

```
In [9]: del dict_alph['a']

In [10]: dict_alph
Out[10]: {'b': 2, 'c': 30, 'd': 4, 'e': 5, 'f': 6}
```

Attempting to Access a Nonexistent Key

```
In [11]: dict_alph['a']
---------------------------------------------------------------------------

KeyError                                  Traceback (most recent
     call last)
Cell In[11], line 1
----> 1 dict_alph['a']

KeyError: 'a'
```

Dictionaries

- You can prevent this error by using dictionary method get, which normally returns its argument's corresponding value. If that key is not found, get returns None.
- IPython does not display anything when None is returned. If you specify a second argument to get, it returns that value if the key is not found:

```
In [12]: dict_alph.get('a')

In [13]: dict_alph.get('a', 'Key is Not Found')
Out[13]: 'Key is Not Found'
```

- Testing Whether a Dictionary Contains a Specified Key

```
In [14]: 'a' in dict_alph
Out[14]: False

In [15]: 'b' in dict_alph
Out[15]: True
```

Dictionaries

- Dictionary Methods keys and values
  - keys and values methods can be used to iterate through only a dictionary's keys or values, respectively:

```
In [21]:  months = {'January': 1, 'February': 2, 'March': 3}

In [22]: for m in months.keys():
    ...:     print(m, end= '  ')
    ...:
January  February  March
In [23]: for n in months.values():
    ...:     print(n, end='  ')
    ...:
1  2  3
```

- Dictionary Views
  - Dictionary methods items, keys and values each return a view of a dictionary's data.
  - When you iterate over a view, it "sees" the dictionary's current contents—it does not have its own copy of the data.

Dictionaries

- Converting Dictionary Keys, Values and Key–Value Pairs to Lists
  - To obtain such a list, pass the view returned by keys, values or items to the built-in list function.
  - Modifying these lists does not modify the corresponding dictionary.
- Processing Keys in Sorted Order
  - To process keys in sorted order, you can use built-in function sorted.

Problem-I: Dictionary of Student Grades

You are tasked with designing a Python program to calculate and display individual student averages and the class average based on exam grades. The program should:

Represent the grade book using a dictionary, where:

- Each key is a student's name (string)
- Each value is a list of integers representing the student's exam grades

Implement functionality to:

- Iterate through the dictionary, unpacking each student's name and their list of grades.
- Compute each student's total and average grade.
- Display each student's average with two decimal places.
- Keep track of the total and count of all grades across all students.
- Compute and display the class average with two decimal places

Problem-II: Word Count

You are tasked with writing a Python program that counts the occurrences of each unique word in a given string and displays the results in a formatted table. The program should also determine and print the total number of unique words.

Dictionaries

- Python Standard Library Module collections
  - The module collections contains the type Counter, which receives an iterable and summarizes its elements.

```
import string
from collections import Counter

# Sample text
text = """Bumrah, India's stand-in skipper in the absence of Rohit Sharma,
led the side from front and accounted for eight wickets in both the
innings combined. He was brilliantly assisted by Harshit Rana, who was
playing his maiden Test. The youngster scalped a total of four wickets
(three in 1st and one in 2nd innings), while senior pro Mohammed Siraj
picked five wickets (two in 1st and three in 2nd innings). All-rounder
Nitesh Reddy, who also made his Test debut, made his presence felt in
both departments of the game. He amassed a total of 79 runs (41 in 1st
innings and 38* in the second). With the ball, Reddy removed Mitchell
Marsh on 47 in the second innings, his first scalp in Test cricket."""

# Step 1: Clean the text
# - Use a filter to retain only alphanumeric characters and spaces.
# - Convert all characters to lowercase for uniformity.
# - Split the text into individual words.
cleaned_text = ''.join(filter(lambda ch: ch.isalnum() or ch.isspace(),
     text)).lower().split()

# Step 2: Count the frequency of each word
# - Use the Counter class from collections to create a dictionary-like
     object
#   where keys are words and values are their counts.
counter = Counter(cleaned_text)
```

Dictionaries

```
# Step 3: Display the word counts in a formatted table
# - Print a header with aligned columns.
# - Sort the words alphabetically and display their counts.
print(f"{'Word':<12}\t {'Count':>5}")
for word, count in sorted(counter.items()):
    print(f"{word:<12}\t {count:>5}")
```

Dictionaries

- Dictionary Method update
  - You may insert and update key–value pairs using dictionary method update.
  - It supports:
    a) A dictionary of key-value pairs.
    b) Keyword arguments.
    c) An iterable of key-value tuples.

```
# 1. Using a dictionary
country_codes = {}
country_codes.update({'Australia': 'ar', 'India': 'in'})
print(country_codes)

# 2. Using keyword arguments
country_codes.update(Australia='au')
print(country_codes)

# 3. Using a list of tuples
country_codes.update([('Canada', 'ca'), ('USA', 'us')])
print(country_codes)
```

Dictionaries

- Dictionary Comprehensions
  - **What are Dictionary Comprehensions?**
    - ⋆ A concise way to create dictionaries.
    - ⋆ Often used to map one dictionary to another.
    - ⋆ Syntax: {`key:value for key, value in iterable`}

```
# Reversing keys and values
In [1]: months = {'Jan':1, 'Feb':2, 'Mar':3}
In [2]: months1 = {num:name for name,num in months.items()}
In [3]: months1
Out[3]: {1: 'Jan', 2: 'Feb', 3: 'Mar'}

# Handling duplicate values
In [4]: months2 = {'Jan':1, 'Feb':2, 'Mar':2}
In [5]: months3 = {num:name for name, num in months2.items()}
In [6]: months3
Out[6]: {1: 'Jan', 2: 'Mar'}

# Transforming values
In [7]: grades = {'Sue': [87,56,96], 'Bob':[56, 88, 95]}
In [8]: grades1 = {name: sum(g)/len(g) for name, g in grades.items()}
In [9]: grades1
Out[9]: {'Sue': 79.66666666666667, 'Bob': 79.66666666666667}
```

Dictionaries

- Dictionary Comprehensions
  - **Key Points:**
    - ⋆ Duplicate keys overwrite existing values.
    - ⋆ Useful for transforming keys or values efficiently.
    - ⋆ Can handle complex operations, such as computing averages.

Sets

- A set is an unordered collection of unique values.
- Sets may contain only immutable objects, like strings, ints, floats and tuples that contain only immutable elements.
- Though sets are iterable, they are not sequences and do not support indexing and slicing with square brackets, [ ].
- Dictionaries also do not support slicing.
- Creating a Set with Curly Braces

```
In [1]: n_set = {1,2,3,4,5}

In [2]: n_set
Out[2]: {1, 2, 2, 3, 4, 5}
```

  - Notice that the duplicate int 2 was ignored (without causing an error).
  - An important use of sets is duplicate elimination, which is **automatic** when creating a set.

Sets

- The sets values are not displayed in the same order as they are listed while creating them.
- Sets are **unordered**.
- Why Only Immutable Objects in Sets?
  - **Hashing Requirement:** Sets use a hash table for efficient membership checks and operations.
    For an object to be used as a key in the hash table:
    - Its hash must not change over time.
    - This ensures consistency in locating the object within the set.
  - **Mutable Objects Break Consistency:** If a mutable object (e.g., a list) were allowed in a set, its value could change, altering its hash.
    This would lead to:
    - Errors in locating the object in the set.
    - Potentially breaking the uniqueness guarantee of the set.

Sets

- ## Determining a Set's Length

```
In [6]: colors = {'red', 'blue', 'green', 'yellow', 'orange', 'pink'}

In [7]: len(colors)
Out[7]: 6
```

- ## Checking Whether a Value Is in a Set

```
In [8]: 'red' in colors
Out[8]: True

In [9]: 'purple' in colors
Out[9]: False
```

- ## Iterating Through a Set

```
In [11]: for color in colors:
    ...:     print(color.upper(), end=' ')
    ...:
YELLOW GREEN BLUE ORANGE PINK RED
```

- ## Creating a Set with the Built-In set Function

```
In [12]: numbers = list(range(10))+ [7,8,9,10,11,12]

In [13]: numbers
Out[13]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 7, 8, 9, 10, 11, 12]

In [14]: set(numbers)
Out[14]: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
```

Sets

- Creating a Set with the Built-In set Function
  - If you need to create an empty set, you must use the set function with empty parentheses, rather than empty braces, {}, which represent an empty dictionary.

```
In [15]: es = set()

In [16]: es
Out[16]: set()
```

- Frozenset: An Immutable Set Type
  - Sets are **mutable**—you can add and remove elements, but *set elements* must be *immutable*.
  - Therefore, a set cannot have other sets as elements.
  - A frozenset is an immutable set—it cannot be modified after you create it, so a set can contain frozensets as elements.
  - The built-in function frozenset creates a frozenset from any iterable.

Sets

- Comparing Sets
  - ⋆ Equality (==) and Inequality (!=)

```
In [19]: {1,2,3}=={2,1,3}
Out[19]: True

In [20]: {1,2,3}!={2,3,4}
Out[20]: True
```

  - ⋆ The $<$ operator tests whether the set to its left is a proper subset of the one to its right.

```
In [21]: {1,2,3}<{2,1,4,5}
Out[21]: False

In [22]: {1,2}<{2,1,3}
Out[22]: True
```

Sets

- Comparing Sets
  - ⋆ The $<=$ operator tests whether the set to its left is a improper subset of the one to its right.

```
In [23]: {1,2}<={2,1}
Out[23]: True

In [24]: {1,2}<={1,2,3}
Out[24]: True
```

  - ⋆ The $>=$ operator tests whether the set to its left is an improper superset of the one to its right
  - ⋆ You may also check for an improper subset with the set method issubset.
  - ⋆ You may also check for an improper superset with the set method issuperset

```
In [26]: {1,2,5}.issuperset({2,5})
Out[26]: True
```

  - ⋆ The argument to issubset or issuperset can be any iterable.

Sets

- Mathematical Set Operations
  - Union: The union of two sets is a set consisting of all the unique elements from both sets. You can calculate the union with the | operator or with the set type's union method.

```
In [27]: {1,2,3} | {3,4}
Out[27]: {1, 2, 3, 4}

In [28]: {1,2,3}.union({3,4})
Out[28]: {1, 2, 3, 4}
```

  - Intersection: The intersection of two sets is a set consisting of all the unique elements that the two sets have in common. You can calculate the intersection with the operator or with the set type's intersection method.

Sets

- Mathematical Set Operations

```
In [29]: {1,2,3}&{3,4}
Out[29]: {3}

In [30]: {1,2,3}.intersection({3,4})
Out[30]: {3}
```

- Difference: The difference between two sets is a set consisting of the elements in the left operand that are not in the right operand. You can calculate the difference with the - operator or with the set type's difference method.

```
In [31]: {1,2,3}-{3,4,5}
Out[31]: {1, 2}

In [32]: {1,2,3}.difference({3,4,5})
Out[32]: {1, 2}
```

Sets

- Mathematical Set Operations
  - Symmetric Difference: The symmetric difference between two sets is a set consisting of the elements of both sets that are not in common with one another. You can calculate the symmetric difference with the ^ operator or with the set type's `symmetric_difference` method.

```
In [33]: {1,2,3}^{2,3,4}
Out[33]: {1, 4}

In [34]: {1,2,3}.symmetric_difference({2,3,4})
Out[34]: {1, 4}
```

  - Disjoint: Two sets are disjoint if they do not have any common elements. You can determine this with the set type's isdisjoint method.

```
In [36]: {1,2,3}.isdisjoint({4,5})
Out[36]: True
```

Sets

- Mutable Set Operators and Methods:
  - Mutable Mathematical Set Operations:
    - Python provides operators and methods to modify sets directly instead of creating new ones.
    - **Union Augmented Assignment** $(|=)$:

```
In [41]: numbers = {1,4,6}
In [42]: numbers |={1,5}
In [43]: numbers
Out[43]: {1, 4, 5, 6}
```

    - **Intersection Augmented Assignment** $(\&=)$**:** Modifies the set to contain only elements present in both sets.
    - **Difference Augmented Assignment (-=):** Removes elements from the set that are present in another set.
    - **Symmetric Difference Augmented Assignment (≙):** Updates the set to contain elements in either set but not both.

Sets

- Mutable Set Operators and Methods:
  - Mutable Mathematical Set Operations:
    Methods with Iterables:
    - **update (Union):**

```
In [44]: numbers.update(range(10))
In [45]: numbers
Out[45]: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

    - `intersection_update`, `difference_update`, `symmetric_differenc_update`: Perform their respective operations with iterable arguments.
  - Methods for Adding and Removing Elements:
    - `add(element)`: Adds an element to the set if it doesn't exist.

Sets

- Mutable Set Operators and Methods:
  - Methods for Adding and Removing Elements:
    - remove(element): Removes an element; raises a KeyError if the element doesn't exist

```
In [44]: numbers.update(range(10))
In [45]: numbers
Out[45]: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

    - `remove(element)`: Removes an element; raises a `KeyError` if the element doesn't exist
    - `discard(element)`: Removes an element; does nothing if the element doesn't exist.
    - `pop()`: Removes and returns an arbitrary element. Raises a `KeyError` if the set is empty.
    - `clear()`: Removes all elements from the set.