**Module 10. Dictionaries and Sets**

**Learning Objectives**

* Understand the creation and manipulation of dictionaries in Python for data association.
* Utilize dictionary methods to access, add, retrieve, and remove items efficiently.
* Learn the creation and manipulation of sets in Python to handle unique values.
* Apply various set methods to perform operations like union, intersection, and difference.
* Use the pickle module to serialize and deserialize Python objects for data storage and transfer.

1. **Dictionaries**

What does a real dictionary you find in a library contain? A traditional dictionary has a definition for each term. In other words, there is a definition associated with each term.

A Python dictionary data type (and some particular data structures such as the map) also stores associations, but the types of data for 'what' is associated with 'what' can be any valid object type. The Python dictionary associates a “value” with each “key,” meaning that the key is used to 'look up' the associated value.

In Python terms, a dictionary contains a group of “items,” where each item contains a key and the associated value. Some people refer to this kind of item as a “mapping.”

**Characteristics of Python Dictionaries**

* **Unordered Collections**:
  + Dictionaries are unordered collections of key-value pairs. Unlike lists or tuples, dictionaries do not maintain any order for the elements.
  + In Python 3.7 and later, dictionaries maintain the insertion order, although this behavior should not be relied upon for algorithmic complexity guarantees.
* **Key-Value Pairs**:
  + Each item in a dictionary is a pair consisting of a key and a value. Keys must be unique and immutable (e.g., strings, numbers, or tuples), while values can be of any data type and can be duplicated.
* **Mutable**:
  + Dictionaries are mutable, meaning you can change, add, or remove key-value pairs after the dictionary has been created.
* **Fast Lookups**:
  + Dictionaries are implemented using hash tables, which allows for fast lookup, insertion, and deletion operations. The average time complexity for these operations is O(1).
* **Dynamic**:
  + Dictionaries can grow and shrink as needed, just like lists.
* **Built-in Methods**:
  + Dictionaries come with several built-in methods:
* **dict.keys()**: Returns a view object that displays a list of all the keys.
* **dict.values()**: Returns a view object that displays a list of all the values.
* **dict.items()**: Returns a view object that displays a list of tuples, each containing a key-value pair.
* **dict.get(key[, default])**: Returns the value for the specified key if the key is in the dictionary; otherwise, returns the default value.
* **dict.update([other])**: Updates the dictionary with elements from another dictionary or from an iterable of key-value pairs.
* **dict.pop(key[, default])**: Removes the specified key and returns the corresponding value. If the key is not found, returns the default value.
* **dict.popitem()**: Removes and returns an arbitrary key-value pair as a tuple.

**Creating a Dictionary**

A dictionary can be created in Python by:

* Calling the ***dict*** constructor function:

|  |
| --- |
| d = dict([["roses", "red"], ["violets", "blue"]]) |

* Declaring a dictionary using an initializer with the proper syntax:

|  |
| --- |
| d = {"alice": "bob", "spam": "eggs"} |

Although there are a few variations of the constructor method, you must pass in the data for the keys and values separately. In the constructor example above, the data is passed as a list of lists, where each inner list contains a key and its associated value.

**Accessing Dictionary Items**

Assuming your dictionary is stored in a variable and contains the associations (items) already, you can 'look up' the value for a key by indexing the dictionary.

|  |
| --- |
| d = {"alice": "bob", "spam": "eggs"}  d["alice"] # Output: 'bob' |

**Adding Items to a Dictionary**

Adding an item to an existing dictionary can be done by using a new index and setting the new value.

|  |
| --- |
| d["alice"] = "bob" |

**Retrieving Items from a Dictionary**

You can use the ***get*** method to retrieve the value associated with a key.

|  |
| --- |
| d.get("alice") # Output: 'bob' |

**Removing Items from a Dictionary**

Removing an item from the dictionary can be done by calling the ***pop*** method. This will remove the item and any association between “bob” and the value.

|  |
| --- |
| d.pop("bob") |

Another way to remove an item is by using the ***del*** keyword. Using the del keyword is another more universal way to remove an object or part of one, and of course it applies to a dictionary:

|  |
| --- |
| del d["alice"] |

This removes the association between “alice” and its value.

**Other Dictionary Operations**

Though the most common tasks with a dictionary are most likely adding or removing items and retrieving the value for a key, a good Python programmer will exercise all of its capabilities.

Other dictionary class functions like ***keys*** and ***values*** will produce the current contents of the keys or the values in the dictionary.

* **Keys and Values**: You can retrieve the keys and values of a dictionary using the ***keys*** and ***values*** methods.

|  |
| --- |
| d = {"alice": "bob", "spam": "eggs"}  d.keys() # Output: dict\_keys(['alice', 'spam'])  list(d.keys()) # Output: ['alice', 'spam']  d.values() # Output: dict\_values(['alice', 'spam'])  list(d.values()) # Output: ['bob', 'eggs'] |

Ultimately, the primary use of a dictionary is for storing associations and retrieving them. Do you think a Python dictionary would be useful for tracking the number of occurrences of each word in a document? Take a moment to consider your answer before continuing.

Since a particular value, an integer count, would be associated with each word, which is a string, the dictionary is a highly effective data type and data structure for the word count problem.

In reality, there are endless applications for the storage and processing of associations, as there are numerous interpretations of data that can exhibit associations.

**Example Usage**

|  |
| --- |
| # Creating a dictionary  student\_scores = {  "Alice": 85,  "Bob": 90,  "Charlie": 78  }  # Accessing values by key  print(student\_scores["Alice"]) # Output: 85  # Adding a new key-value pair  student\_scores["David"] = 92  # Updating an existing key-value pair  student\_scores["Alice"] = 88  # Removing a key-value pair  del student\_scores["Charlie"]  # Iterating over keys  for student in student\_scores:  print(student, student\_scores[student])  # Using built-in methods  keys = student\_scores.keys()  values = student\_scores.values()  items = student\_scores.items()  print(keys) # Output: dict\_keys(['Alice', 'Bob', 'David'])  print(values) # Output: dict\_values([88, 90, 92])  print(items) # Output: dict\_items([('Alice', 88), ('Bob', 90), ('David', 92)]) |

**Example 1: Word Count**

A Python dictionary can be useful in tracking the number of occurrences of each word in a document. Since a particular value (an integer count) would be associated with each word (a string), the dictionary is a highly effective data type and data structure for this problem.

def main():

    # Example text to be analyzed

    example\_text = """

    Python is an easy to learn, powerful programming language.

    It has efficient high-level data structures and a simple

    but effective approach to object-oriented programming.

    Python’s elegant syntax and dynamic typing,

    together with its interpreted nature, make it an ideal language

    for scripting and rapid application development

    in many areas on most platforms.

    """

    # Declare a dictionary to store word counts

    word\_counts = {}

    # Split the text into lines

    # (although it's just one block of text here)

    for line in example\_text.splitlines():

        # Split each line into words

        for word in line.split():

            # Remove punctuation and convert to lowercase

            word = word.strip(".,").lower()

            # Increase the count for the word in the dictionary

            if word not in word\_counts:

                # Set the count to 1 for a new word

                word\_counts[word] = 1

            else:

                # Increase the count for an existing word

                word\_counts[word] += 1

    # Print out all word counts

    for word in word\_counts:

        print(f"{word} occurs {word\_counts[word]} times")

if \_\_name\_\_=="\_\_main\_\_":

    main()

1. **Sets: In or Out, No Duplicates**

In programming, different data structures provide various tools and techniques to organize and manage data effectively. One such data structure is the set. Sets are designed to hold an unordered collection of unique values. They automatically exclude any duplicate values, making them ideal for certain types of data processing tasks.

**Why Use Sets?**

Sets are particularly useful when dealing with membership-related tasks, such as:

* Checking if a specific value is in a set.
* Determining common values between two sets.

**Characteristics of Python Sets**

* **Unordered Collections**:
  + Sets are unordered collections of unique elements. This means that the order of elements in a set is not guaranteed and can change.
* **Unique Elements**:
  + All elements in a set are unique. If you try to add a duplicate element, it will be ignored.
* **Mutable**:
  + Sets are mutable, allowing you to add, remove, and update elements after the set has been created.
* **Immutable Counterpart - Frozenset**:
  + Python also provides a frozenset, which is an immutable version of a set. Once created, elements cannot be added or removed from a frozenset.
* **Built-in Methods**:
  + Sets come with several built-in methods for common operations, such as:
  + **set.add(elem)**: Adds an element to the set.
  + **set.update(iterable)**: Updates the set with elements from an iterable.
  + **set.remove(elem)**: Removes an element from the set; raises a KeyError if not found.
  + **set.discard(elem)**: Removes an element if present; does nothing if not found.
  + **set.pop()**: Removes and returns an arbitrary element from the set; raises a KeyError if the set is empty.
  + **set.clear()**: Removes all elements from the set.
  + **set.union(\*others)**: Returns a new set with elements from the set and all others.
  + **set.intersection(\*others)**: Returns a new set with elements common to the set and all others.
  + **set.difference(\*others)**: Returns a new set with elements in the set that are not in the others.
  + **set.symmetric\_difference(other)**: Returns a new set with elements in either the set or other but not both.

You can access detailed documentation about these methods by using the help function in Python interactive mode.

|  |
| --- |
| >>> help(set.add)  Help on method\_descriptor:  add(...)  Add an element to a set.  This has no effect if the element is already present. |

**Examples of Data in Sets**

The data in a set can represent anything, from financial information to social media posts, or even data used to analyze severe storms. The flexibility of sets in handling various data types makes them universally useful.

**Sets in Python**

Python provides built-in support for sets, making them easy to use in your programs. Sets in Python are implemented as an object-oriented class, meaning they come with a set of predefined functions (or methods) that you can use to manipulate the set.

**Creating a Set in Python**

To create a set in Python, you can use the set constructor function. Here’s an example:

|  |
| --- |
| set1 = set([1, 2, 4]) |

In this example, ***set1*** is a set containing the values 1, 2, and 4.

Another way to create a set is by using curly braces {}. Here’s how:

|  |
| --- |
| set1 = {3, 5, 6} |

This also creates a set containing the values 3, 5, and 6.

**Example Usage**

|  |
| --- |
| # Creating a set  fruits = {"apple", "banana", "cherry"}  # Adding an element  fruits.add("orange")  # Removing an element  fruits.remove("banana")  # Checking for membership  print("apple" in fruits) # Output: True  # Iterating over a set  for fruit in fruits:  print(fruit)  # Using set operations  set1 = {1, 2, 3}  set2 = {3, 4, 5}  union\_set = set1.union(set2) # {1, 2, 3, 4, 5}  intersection\_set = set1.intersection(set2) # {3}  difference\_set = set1.difference(set2) # {1, 2}  symmetric\_difference\_set = set1.symmetric\_difference(set2) # {1, 2, 4, 5} |

1. **Serializing Objects – The pickle Module**

The ***pickle*** module in Python is used for serializing and deserializing Python objects. Serialization, also known as pickling, is the process of converting a Python object into a byte stream, which can be stored in a file or transmitted over a network. Deserialization, or unpickling, is the reverse process where the byte stream is converted back into a Python object.

**Steps to Use the pickle Module for Serializing Objects**

* **Import the pickle Module**: You need to import the ***pickle*** module to use its functions.

import pickle

* **Open a File for Binary Writing**: You need to open a file in binary mode for writing (***wb***). This file will store the serialized byte stream.

with open('data.pkl', 'wb') as file:

    # Perform pickling operations here

* **Pickle the Object and Write it to the File**: Use the ***pickle.dump*** method to serialize the object and write it to the specified file.

with open('data.pkl', 'wb') as file:

    pickle.dump(my\_object, file)

Here, ***my\_object*** can be any Python object, such as a dictionary or a set.

* **Close the File**: If you are using the ***with*** statement, the file will be automatically closed. Otherwise, you should explicitly close the file.

file.close()

**Example: Serializing a Dictionary and a Set**

Here's a complete example that demonstrates how to pickle a dictionary and a set:

import pickle

# Example dictionary and set

my\_dict = {'name': 'Alice', 'age': 30, 'city': 'New York'}

my\_set = {'apple', 'banana', 'cherry'}

# Open a file for binary writing

with open('data.pkl', 'wb') as file:

    # Pickle the dictionary

    pickle.dump(my\_dict, file)

    # Pickle the set

    pickle.dump(my\_set, file)

**Steps to Deserialize the Objects**

* **Open the File for Binary Reading**: Open the file in binary mode for reading (***rb***).

with open('data.pkl', 'rb') as file:

    # Perform unpickling operations here

* **Unpickle the Objects**: Use the ***pickle.load*** method to deserialize the objects from the file.

with open('data.pkl', 'rb') as file:

    loaded\_dict = pickle.load(file)

    loaded\_set = pickle.load(file)

**Example: Deserializing the Dictionary and Set**

This example code shows how to unpickle a dictionary and a set:

import pickle

# Open the file for binary reading

with open('data.pkl', 'rb') as file:

    # Unpickle the dictionary

    loaded\_dict = pickle.load(file)

    # Unpickle the set

    loaded\_set = pickle.load(file)

# Display the loaded objects

print(loaded\_dict)

print(loaded\_set)

**Putting it all together**

import pickle

def main():

    # Example dictionary and set

    my\_dict = {'name': 'Alice', 'age': 30, 'city': 'New York'}

    my\_set = {'apple', 'banana', 'cherry'}

    # Open a file for binary writing

    with open('data.pkl', 'wb') as file:

        # Pickle the dictionary

        pickle.dump(my\_dict, file)

        # Pickle the set

        pickle.dump(my\_set, file)

    # Open the file for binary reading

    with open('data.pkl', 'rb') as file:

        # Unpickle the dictionary

        loaded\_dict = pickle.load(file)

        # Unpickle the set

        loaded\_set = pickle.load(file)

    # Display the loaded objects

    print(loaded\_dict)

    print(loaded\_set)

if \_\_name\_\_=="\_\_main\_\_":

    main()

Output will be:

{'name': 'Alice', 'age': 30, 'city': 'New York'}

{'banana', 'cherry', 'apple'}

**Summary**

1. A Python dictionary stores associations between keys and values, allowing efficient data retrieval.
2. Dictionaries can be created using the ***dict*** constructor or by declaring a dictionary with key-value pairs.
3. Accessing dictionary items is done by indexing the dictionary with the key.
4. Adding an item to a dictionary involves assigning a value to a new key.
5. The ***get*** method retrieves the value associated with a key in a dictionary.
6. Items can be removed from a dictionary using the ***pop*** method or the ***del*** keyword.
7. The ***keys*** method retrieves all the keys in a dictionary, while the ***values*** method retrieves all the values.
8. Python dictionaries are effective for tasks like word count, where each word is associated with its count.
9. Sets in Python hold an unordered collection of unique values, automatically excluding duplicates.
10. Sets are useful for membership-related tasks like checking if a value is in a set or finding common values.
11. A set can be created using the ***set*** constructor or by using curly braces {}.
12. Common set methods include ***add()***, ***remove()***, ***intersection()***, and ***union()***.
13. The ***pickle*** module in Python is used for serializing and deserializing Python objects.
14. Serialization (pickling) converts a Python object into a byte stream for storage or transmission.
15. Deserialization (unpickling) converts a byte stream back into a Python object.

**Programming Exercises**

1. **Dictionary of Synonyms**

Write a program that creates a dictionary where the keys are words and the values are lists of synonyms. Allow the user to input a word and display its synonyms from the dictionary.

1. **Country Quiz**

Write a program that creates a dictionary with countries as keys and their capitals as values. Randomly quiz the user on country capitals, keeping track of correct and incorrect responses.

1. **Phonebook Application**

Write a program that allows users to add, remove, and look up contacts in a phonebook. Use a dictionary where the keys are contact names and the values are phone numbers.

1. **Student Grades**

Write a program that creates a dictionary with student names as keys and their grades as values. Allow the user to input student names and display their grades, or add new students and grades.

1. **Letter Frequency Counter**

Write a program that reads a text file and counts the frequency of each letter (ignoring case). Use a dictionary where the keys are letters and the values are their frequencies.

1. **Inventory System**

Write a program that manages an inventory of products using a dictionary. The keys are product names and the values are the quantities in stock. Allow the user to add, remove, and update product quantities.

1. **Anagram Finder**

Write a program that reads a list of words from a file and creates a dictionary where the keys are sorted strings of letters and the values are lists of words that are anagrams of those letters.

1. **Character Replacement Encryption**

Write a program that uses a dictionary to replace each character in a string with a corresponding character. Read the input string from a file and write the encrypted string to another file.

1. **Common Words in Texts**

Write a program that reads two text files and uses sets to find and display the common words between the two files.

1. **Set Operations on Names**

Write a program that reads two lists of names from separate files and stores them in sets. Perform set operations (union, intersection, difference) on these sets and display the results.