MODULE :3

CHAPTER 1: Dictionaries

Dictionaries are used to store data values in key:value pairs.

A dictionary is a collection which is ordered*, changeable and do not allow duplicates.

Example 1:

```
thisdict =
  { "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
```

Duplicates Not Allowed

Dictionaries cannot have two items with the same key

Example 2:

```
thisdict =
  { "brand": "Ford",
  "model": "Mustang",
  "year": 1964,
  "year": 2020
}
print(thisdict)
```

Dictionary Items - Data Types

The values in dictionary items can be of any data type

Example 3:

```
thisdict = { "brand":
  "Ford", "electric":
  False, "year":
  1964,
  "colors": ["red", "white", "blue"]
}
```

Accessing Items

You can access the items of a dictionary by referring to its key name, inside square brackets

Example 4:

```
thisdict =
  { "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = thisdict["model"]
```

There is also a method called get() that will give you the same result

```
x = thisdict.get("model")
```

Get Keys

The keys() method will return a list of all the keys in the dictionary.

The list of the keys is a *view* of the dictionary, meaning that any changes done to the dictionary will be reflected in the keys list

```
x = thisdict.keys()
```

Example 5:

```
car = { "brand":
"Ford",
"model": "Mustang",
"year": 1964
}

x = car.keys()
print(x) #before the change

car["color"] = "white"
print(x) #after the change
```

Get Values

The values() method will return a list of all the values in the dictionary.

The list of the values is a *view* of the dictionary, meaning that any changes done to the dictionary will be reflected in the values list.

```
x = thisdict.values()
```

Example 6:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = car.values()
print(x) #before the change

car["year"] = 2020 print(x)
#after the change
```

Add a new item to the original dictionary, and see that the values list gets updated as well

Example 7:

```
car = { "brand":
"Ford",
"model": "Mustang",
"year": 1964
}
x = car.values()
print(x) #before the change

car["color"] = "red"
print(x) #after the change
```

Check if Key Exists

To determine if a specified key is present in a dictionary use the in keyword

Example 8:

```
thisdict =
  { "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
if "model" in thisdict:
  print("Yes, 'model' is one of the keys in the thisdict
dictionary")
```

Change Values

You can change the value of a specific item by referring to its key name

Example 9:

```
thisdict =
  { "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["year"] = 2018

print(thisdict)
```

Update Dictionary

The update() method will update the dictionary with the items from the given argument.

The argument must be a dictionary, or an iterable object with key:value pairs

Example 10:

```
thisdict =
  { "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.update({"year": 2020})

print(thisdict)
```

Adding Items

Adding an item to the dictionary is done by using a new index key and assigning a value to it

Example 11:

```
thisdict =
  { "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["color"] = "red"
print(thisdict)
```

Removing Items

There are several methods to remove items from a dictionary
The pop() method removes the item with the specified key name

Example 12:

```
thisdict =
  { "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.pop("model")
print(thisdict)
```

The popitem() method removes the last inserted item (in versions before 3.7, a random item is removed instead)

Example 13:

```
thisdict =
  { "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.popitem()
print(thisdict)
```

The del keyword removes the item with the specified key name

Example 14:

```
thisdict =
  { "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict["model"]
print(thisdict)
```

Aliasing in Dictionary

Aliasing means two variables refer to the same dictionary object in memory. So, if one is changed — the other also changes.

```
dict1 = {'a': 1, 'b': 2}
dict2 = dict1     # aliasing (no new copy created)
dict2['a'] = 100
print(dict1)   # {'a': 100, 'b': 2}
print(dict2)   # {'a': 100, 'b': 2}
```

Copying in Dictionary

To avoid aliasing, we can create a copy of the dictionary.

Using copy() method.

```
dict1 = {'a': 1, 'b': 2}
dict2 = dict1.copy()
# makes a new, independent dictionary

dict2['a'] = 100

print(dict1)   # {'a': 1, 'b': 2}
print(dict2)   # {'a': 100, 'b': 2}
```

Numpy is an open-source library in Python that provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays.

- Instead of just dealing with single values like regular Python lists, NumPy lets you work with whole collections of numbers super efficiently — think of it as your toolkit for doing math with arrays. It's lightning fast because it uses optimized C code under the hood.
- **Key Features**:
  - **Efficient Storage**: Numpy arrays use less memory compared to Python lists.
  - **Performance**: Numpy operations are implemented in C and optimized for performance, allowing for faster computations.
  - **Interoperability**: Works seamlessly with many other scientific libraries, including **SciPy**, **Pandas**, and **Matplotlib**.

Creation of Arrays

- **Using `np.array()`** : Convert Python lists or tuples to Numpy arrays.

  Example:

  ```
  import numpy as np
  list_a = [1, 2, 3]
  array_a = np.array(list_a)
  print(array_a)  # Output: [1 2 3]
  ```

2. Shape

The shape of a NumPy array tells you its dimensions — how many rows, columns, or layers it has.

You can check an array's shape with .shape, and you can even change it (reshape) if the total number of elements matches.

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr.shape)
# Output: (2, 3) — 2 rows, 3 columns
```

- **Reshape**: Change the shape of an array without changing its data.

  Example:

  ```
  a = np.arange(6)  # [0, 1, 2, 3, 4, 5]
  b = a.reshape((2, 3))  # Reshaping to 2 rows, 3
  columns
  ```

- **Flattening**: Convert a multi-dimensional array into a one-dimensional array using `flatten()` or `ravel()`.

Example:

```
c = b.flatten()  # Output: [0 1 2 3 4 5]
```

- **Transposing**: Swap rows with columns using `.T`.

  Example:

  ```
  transposed = b.T  # Transpose of b
  ```

3. Slicing

Slicing is how you cut out parts of your array, similar to how you slice lists in Python but more powerful.

arr[1:4] grabs elements from index 1 up to (but not including) 4.

You can slice multi-dimensional arrays too: arr[0:2, 1:3] means take rows 0 and 1, and columns 1 and 2.

Examples:

```
b = np.array([[1, 2, 3], [4, 5, 6]])
print(b[0:2, 1:3])
# Output: [[2 3]
#          [5 6]]

arr = np.array([0, 1, 2, 3, 4, 5])
arr[2:5] = 99
print(arr)
# [ 0  1 99 99 99  5]
```

4. Masking

Masking lets you pick out elements based on conditions.

Suppose you want all numbers greater than 10 from an array. You create a mask like arr > 10, which is a True/False array.

Then do arr[arr > 10] to get just the elements where the mask is True.Great for filtering data without loops.

Example:

```
arr = np.array([3, 6, 9, 12, 15])
```

```
arr[arr % 3 == 0] = -1  # Replace multiples of 3 with -1

print(arr)

# [-1 -1 -1 -1 -1]
```

5. Broadcasting

- **Broadcasting** allows operations on arrays of different shapes. Numpy automatically expands the smaller array to match the larger array's shape.
- **Example of Broadcasting:**

```
a = np.array([[1], [2], [3]])
b = np.array([1, 2, 3])
c = a + b  # The smaller array is broadcasted to
match the shape of the larger array
# Output:
# [[2 3 4]
#  [3 4 5]
#  [4 5 6]]
```

- **Broadcasting Rules**:
    1. If the arrays have a different number of dimensions, prepend the shape of the smaller array with ones until they have the same number of dimensions.
    2. Compare the shapes of the two arrays element-wise, starting from the trailing dimensions. If the sizes of the dimensions are equal or one of them is 1, broadcasting occurs.
    3. If the sizes are incompatible, an error is raised.

6. dtype

dtype means data type — it tells NumPy what kind of data is stored in the array: integers, floats, booleans, etc.

You can specify it when creating an array, e.g., np.array([1, 2, 3], dtype=float).

Knowing dtype is important because it affects memory usage and what operations you can perform.

Example:

```
array_a = np.array([1, 2, 3], dtype=np.float32)  #
Explicitly set type to float
```

NumPy supports lots of types, like int32, float64, bool, and even complex numbers.

- **Numpy Data Types**:
  - `np.int32`, `np.int64`, `np.float32`, `np.float64`, `np.complex`, `np.bool_`, etc.

Changing dtype

To change the dtype of an existing array, you can use the astype method:

```
import numpy as np
a = np.array([200], dtype='uint8')
a.astype('uint64')
```

CHAPTER 3: File Handling

File handling is an essential concept in programming that allows you to create, read, write, update, and delete files using a programming language. In , file handling is straightforward and provides functions to manipulate files in various ways. Files are used to store data permanently, making it possible to save data for future use, share it between different programs, or process large datasets that can't be stored in memory.

File Modes

When opening a file in , you must specify the mode, which determines how the file will be handled. The common modes are:

- **r (Read)**: Opens the file for reading. If the file does not exist, it raises an error.
- **w (Write)**: Opens the file for writing. If the file already exists, it truncates it (removes all content). If the file does not exist, it creates a new file.
- **a (Append)**: Opens the file for appending. If the file exists, new data is added at the end of the file without truncating it. If the file does not exist, it creates a new file.
- **x (Create)**: Creates a new file. If the file already exists, it raises an error.
- **b (Binary)**: Used with other modes to open files in binary mode (e.g., rb, wb).
- **t (Text)**: Used with other modes to open files in text mode (default).

Writing Our First File

To write data to a file, use the open() function with write mode.

Example:

```
# Open a file for writing
file = open("example.txt", "w")
# Write data into it
file.write("Hello, this is my first file!\n")
file.write("Python file handling is easy.\n")
# Close the file
file.close()
```

Reading a File Line-at-a-Time

To read a file one line at a time, use a for loop or readline().
Example 1: Using for loop

```
file = open("example.txt", "r")

for line in file:
   print(line.strip())  # strip() removes newline characters

file.close()
```

Example 2: Using readline()

```
file = open("example.txt", "r")
line1 = file.readline()
line2 = file.readline()
print(line1)
print(line2)

file.close()
```

Turning a File into a List of Lines
If you want all lines as a list, use readlines() or list comprehension.

```
file = open("example.txt", "r")
lines = file.readlines()
file.close()

print(lines)
```

output:
['Hello, this is my first file!\n', 'Python file handling is easy.\n']

Reading the Whole File at Once
If the file is small, you can read it entirely using read().

```
file = open("example.txt", "r")
content = file.read()
file.close()

print(content)
```

Work with Binary Files: When working with non-text files like images or executables, open them in binary mode using rb, wb, etc.

- To read or write binary files in Python, open the file with 'rb' (read binary) or 'wb' (write binary) modes.
- This lets you handle data exactly as it is, byte by byte.

Example: Writing and reading binary data

```
# Write binary data
```

```
with open('example.bin', 'wb') as f:
    f.write(b'\x00\x01\x02\x03')

# Read binary data
with open('example.bin', 'rb') as f:
    data = f.read()
     print(data)
      # Output: b'\x00\x01\x02\x03'
```

1. Directories

Directories (or folders) organize files. You can use Python's os or pathlib modules to work with directories:

- Create, list, or check directories.
- Navigate the file system programmatically.

Example: Listing all files in a directory

```
import os

files = os.listdir('.')  # List all files/folders in
current directory
print(files)
```

Example: Creating a new directory

```
import os

os.mkdir('new_folder')
print('Directory created!')
```

2. Fetching Something from the Web

To get data from the internet (like downloading a webpage or file), you can use the requests library (needs to be installed).

Example: Fetching a webpage

```
import requests

response = requests.get('https://www.example.com')

print(response.status_code)
# 200 means success
print(response.text[:200])
# print first 200 characters of the page
```

Example: Download and save an image

```
url = 'https://example.com/image.jpg'
response = requests.get(url)

with open('downloaded_image.jpg', 'wb') as f:
    f.write(response.content)

print('Image downloaded!')
```