Module 07: Working with Files

Agenda

- Open File
- Reading file data
- Writing to file
- File Position
- File Attributes
- JSON

Open Files

- **file_object = open(filename, mode = 'r')**
- *modes*:
  - **'r'** - Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
  - **'w'** - Opens a file for writing. If file already exists it will be truncated, if not it will be created.
  - **'a'** - Opens a file for appending. The file pointer is at the end of the file if the file exists. If the file does not exist, it creates a new file for writing.
  - **'r+'** - Opens a file for both reading and writing. The file pointer will be at the beginning of the file
  - **'w+'** - Opens a file for both writing and reading. If file already exists it will be truncated, if not it will be created.

Close Files

- ***file_object*.close()** - to close it and free up any system resources taken up by the file
- ***file_object*.closed** – returns whether the file is already closed (True/False)

- It is good practice to use the **with** keyword when dealing with file objects (context manger)
- This has the advantage that the file is properly closed after its suite finishes, even if an exception is raised on the way.
- It is also much shorter than writing equivalent try-finally blocks

- ```
  with open('output.txt', 'w') as f:
      f.write('Hi there!')
  ```
  ```

Reading data

- *file_object*.**readline()**
  - reads a single line from the file a newline character (\n) is left at the end of the string
  - if f.readline() returns an empty string, the end of the file has been reached

- *file_object*.**readlines( )** – return a list containing the file content lines

```
file = open('somefile.txt', 'r')
print (file.readlines())          #['first line\n', 'second line\n']
```

Reading data – cont’d

- It is possible to read file data with **read** function
- *file_object*.read(size) - reads some quantity of data and returns it as a string:
  - s = f.read(size)
- *size* is an optional numeric argument. When *size* is omitted or negative, the entire contents of the file will be read and returned:
  - whole_data = *file_object*.read() - reads the entire contents of the file

File iterator

- For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code:

```
for line in file_object:
    print (line)
```

Writing to file

- *file_object*.write(string) - writes the contents of *string* to the file

```
with open('out', 'w') as f:
        f.write('This is a test\n')
```

File Position

- **tell()** - returns current position in the file, in bytes from the beginning of the file.
- The ***seek(offset, from=0)*** method changes the current file position.
  - The *offset* argument indicates the number of bytes to be moved.
  - The *from* argument specifies the reference position from where the bytes are to be moved:
    - **io.SEEK_SET (0)** – from the beginning of the file
    - **io.SEEK_CUR (1)** – from current file position
    - **io.SEEK_END (2)** – from the end of the file
  - offset must be zero for SEEK_CUR and SEEK_END in python 3.x

File object attributes

- **file.closed** - Returns true if file is closed, false otherwise.
- **file.mode** - Returns access mode with which file was opened.
- **file.name** - Returns name of the file.

```
fo = open("foo.txt", "w")
print ("Name of the file: ", fo.name )
print ("Closed or not : ", fo.closed )
print ("Opening mode : ", fo.mode)
```

The Output:

"foo.txt“
False
w

JSON

- JSON (JavaScript Object Notation) is a lightweight data-interchange format that is widely used for transmitting and storing data. It is easy for humans to read and write, and it's also easy for machines to parse and generate. JSON is supported by many programming languages, including Python.

- In Python, you can work with JSON using the built-in json module. This module provides functions for encoding (converting Python objects to JSON strings) and decoding (converting JSON strings to Python objects) data.

- Let's start with an example of encoding a Python object to JSON. Consider a dictionary representing a person's information:

JSON Examples

Demo

JSON Examples

In the example above, we use the dumps() function from the json module to convert the person dictionary into a JSON string. The json_data variable stores the resulting JSON string. Notice that the keys and string values are enclosed in double quotes, as per the JSON syntax.

```
import json

person = {
    "name": "John Doe",
    "age": 25,
    "city": "New York"
}

# Encode the dictionary to JSON
json_data = json.dumps(person)

print(json_data)
```

JSON Examples

In this example, we use the loads() function from the json module to convert the JSON string json_data into a Python dictionary. We can then access the values in the dictionary using the corresponding keys.

```
import json

json_data = '{"name": "John Doe", "age": 25, "city": "New York"}'

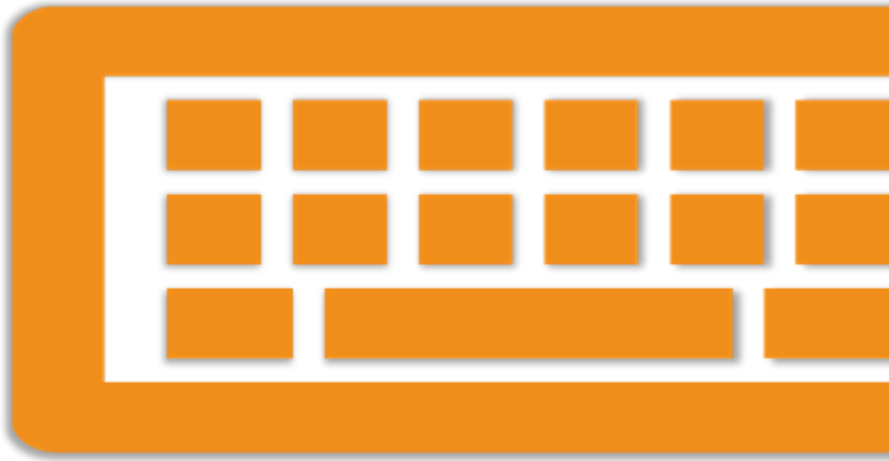
# Decode the JSON string to a dictionary
person = json.loads(json_data)

print(person["name"])
print(person["age"])
print(person["city"])
```

JSON is commonly used for exchanging data between a client and a server or for storing configuration settings. It is a versatile and widely supported format for data serialization.

Labs 01-03

Lab



Questions