Python - Read Files

Reading from a file involves opening the file, reading its contents, and then closing the file to free up system resources. Python provides several methods to read from a file, each suited for different use cases.

Opening a File for Reading

Opening a file is the first step in reading its contents. In Python, you use the open() function to open a file. This function requires at least one argument, the filename, and optionally a mode that specifies the purpose of opening the file.

To open a file for reading, you use the mode **'r'**. This is the default mode, so you can omit it if you only need to read from the file.

Reading a File Using read() Method

The read() method is used to read the contents of a file in Python. It reads the entire content of the file as a single string. This method is particularly useful when you need to process the whole file at once.

Syntax

Following is the basic syntax of the read() method in Python −

```
file_object.read(size)
```

Where,

- **file_object** is the file object returned by the open() function.
- **size** is the number of bytes to read from the file. This parameter is optional. If omitted or set to a negative value, the method reads until the end of the file.

Example

In the following example, we are opening the file "example.txt" in read mode. We then use the read() method to read the entire content of the file −

```
# Open the file in read mode
file = open('example.txt', 'r')
```

```
# Read the entire content of the file
content = file.read()

# Print the content
print(content)

# Close the file
file.close()
```

After executing the above code, we get the following output −

```
welcome to Tutorialspoint.
```

Learn **Python** in-depth with real-world projects through our **Python certification course**. Enroll and become a certified expert to boost your career.

Reading a File Using readline() Method

The readline() method is used to read one line from a file at a time. This method is useful when you need to process a file line by line, especially for large files where reading the entire content at once is not practical.

Syntax

Following is the basic syntax of the readline() method in Python −

```
file_object.readline(size)
```

Where,

- **file_object** is the file object returned by the open() function.
- **size** is an optional parameter specifying the maximum number of bytes to read from the line. If omitted or set to a negative value, the method reads until the end of the line.

Example

In the example below, we are opening the file "example.txt" in read mode. We then use the readline() method to read the first line of the file −

```
# Open the file in read mode
file = open('example.txt', 'r')

# Read the first line of the file
line = file.readline()

# Print the line
print(line)

# Close the file
file.close()
```

Following is the output of the above code −

```
welcome to Tutorialspoint.
```

Reading a File Using readlines() Method

The readlines() method reads all the lines from a file and returns them as a list of strings. Each string in the list represents a single line from the file, including the newline character at the end of each line.

This method is particularly useful when you need to process or analyse all lines of a file at once.

Syntax

Following is the basic syntax of the readlines() method in Python −

```
file_object.readlines(hint)
```

Where,

- **file_object** is the file object returned by the open() function.
- **hint** is an optional parameter that specifies the number of bytes to read. If specified, it reads lines up to the specified bytes, not necessarily reading the entire file.

Example

In this example, we are opening the file "example.txt" in read mode. We then use the readlines() method to read all the lines from the file and return them as a list of strings −

```
# Open the file in read mode
file = open('example.txt', 'r')

# Read all lines from the file
lines = file.readlines()

# Print the lines
for line in lines:
   print(line, end='')

# Close the file
file.close()
```

Output of the above code is as shown below −

```
welcome to Tutorialspoint.
Hi Surya.
How are you?.
```

Using "with" Statement

The "with" statement in Python is used for exception handling. When dealing with files, using the "with" statement ensures that the file is properly closed after reading, even if an exception occurs.

*When you use a **with** statement to open a file, the file is automatically closed at the end of the block, even if an error occurs within the block.*

Example

Following is a simple example of using the **with** statement to open, read, and print the contents of a file −

```
# Using the with statement to open a file
with open('example.txt', 'r') as file:
   content = file.read()
   print(content)
```

We get the output as follows –

```
welcome to Tutorialspoint.
Hi Surya.
How are you?.
```

Reading a File in Binary Mode

By default, read/write operation on a file object are performed on text string data. If we want to handle files of different types, such as media files (mp3), executables (exe), or pictures (jpg), we must open the file in binary mode by adding the 'b' prefix to the read/write mode.

Writing to a Binary File

Assuming that the **test.bin** file has already been written in binary mode –

```
# Open the file in binary write mode
with open('test.bin', 'wb') as f:
   data = b"Hello World"
   f.write(data)
```

Example

To read a binary file, we need to open it in 'rb' mode. The returned value of the read() method is then decoded before printing –

```
# Open the file in binary read mode
with open('test.bin', 'rb') as f:
   data = f.read()
   print(data.decode(encoding='utf-8'))
```

It will produce the following output –

```
Hello World
```

Reading Integer Data From a File

To write integer data to a binary file, the integer object should be converted to bytes using the to_bytes() method.

Writing an Integer to a Binary File

Following is an example on how to write an integer to a binary file −

```
# Convert the integer to bytes and write to a binary file
n = 25
data = n.to_bytes(8, 'big')

with open('test.bin', 'wb') as f:
   f.write(data)
```

Reading an Integer from a Binary File

To read back the integer data from the binary file, convert the output of the read() function back to an integer using the from_bytes() method −

```
# Read the binary data from the file and convert it back to an integer
with open('test.bin', 'rb') as f:
   data = f.read()
   n = int.from_bytes(data, 'big')
   print(n)
```

Reading Float Data From a File

For handling floating-point data in binary files, we need to use the **struct** module from Python's standard library. This module helps convert between Python values and C structs represented as Python bytes objects.

Writing a Float to a Binary File

To write floating-point data to a binary file, we use the struct.pack() method to convert the float into a bytes object −

```
import struct

# Define a floating-point number
x = 23.50

# Pack the float into a binary format
```

```
data = struct.pack('f', x)

# Open the file in binary write mode and write the packed data
with open('test.bin', 'wb') as f:
   f.write(data)
```

Reading Float Numbers from a Binary File

To read floating-point data from a binary file, we use the struct.unpack() method to convert the bytes object back into a float –

```
import struct

# Open the file in binary read mode
with open('test.bin', 'rb') as f:
   # Read the binary data from the file
   data = f.read()

   # Unpack the binary data to retrieve the float
   x = struct.unpack('f', data)[0]

   # Print the float value
   print(x)
```

Reading and Writing to a File Using "r+" Mode

When a file is opened for reading (with 'r' or 'rb'), writing data is not possible unless the file is closed and reopened in a different mode. To perform both read and write operations simultaneously, we add the '+' character to the mode parameter. Using 'w+' or 'r+' mode enables using both write() and read() methods without closing the file.

The File object also supports the seek() function, which allows repositioning the read/write pointer to any desired byte position within the file.

Syntax

Following is the syntax for seek() method –

```
fileObject.seek(offset[, whence])
```

Parameters

- **offset** – This is the position of the read/write pointer within the file.
- **whence** – This is optional and defaults to 0 which means absolute file positioning, other values are 1 which means seek relative to the current position and 2 means seek relative to the file's end.

Example

The following program opens a file in 'r+' mode (read-write mode), seeks a certain position in the file, and reads data from that position –

```
# Open the file in read-write mode
with open("foo.txt", "r+") as fo:
   # Move the read/write pointer to the 10th byte position
   fo.seek(10, 0)

   # Read 3 bytes from the current position
   data = fo.read(3)

   # Print the read data
   print(data)
```

After executing the above code, we get the following output –

```
rat
```

Reading and Writing to a File Simultaneously in Python

When a file is opened for writing (with 'w' or 'a'), it is not possible to read from it, and attempting to do so will throw an **UnsupportedOperation** error.

Similarly, when a file is opened for reading (with 'r' or 'rb'), writing to it is not allowed. To switch between reading and writing, you would typically need to close the file and reopen it in the desired mode.

To perform both read and write operations simultaneously, you can add the '+' character to the mode parameter. Using 'w+' or 'r+' mode enables both write() and read() methods without needing to close the file.

Additionally, the File object supports the seek() function, which allows you to reposition the read/write pointer to any desired byte position within the file.

Example

In this example, we open the file in 'r+' mode and write data to the file. The seek(0) method repositions the pointer to the beginning of the file –

```
# Open the file in read-write mode
with open("foo.txt", "r+") as fo:
   # Write data to the file
   fo.write("This is a rat race")

   # Rewind the pointer to the beginning of the file
   fo.seek(0)

   # Read data from the file
   data = fo.read()
   print(data)
```

Reading a File from Specific Offset

We can set the he file's current position at the specified offset using the seek() method.

- If the file is opened for appending using either 'a' or 'a+', any seek() operations will be undone at the next write.
- If the file is opened only for writing in append mode using 'a', this method is essentially a no-op, but it remains useful for files opened in append mode with reading enabled (mode 'a+').
- If the file is opened in text mode using 't', only offsets returned by tell() are legal. Use of other offsets causes undefined behavior.

Note that not all file objects are seekable.

Example

The following example demonstrates how to use the seek() method to perform simultaneous read/write operations on a file. The file is opened in w+ mode (read-write mode), some data is added, and then the file is read and modified at a specific position –

Open Compiler

```
# Open a file in read-write mode
fo = open("foo.txt", "w+")

# Write initial data to the file
```

```
fo.write("This is a rat race")

# Seek to a specific position in the file
fo.seek(10, 0)

# Read a few bytes from the current position
data = fo.read(3)
print("Data read from position 10:", data)

# Seek back to the same position
fo.seek(10, 0)

# Overwrite the earlier contents with new text
fo.write("cat")

# Rewind to the beginning of the file
fo.seek(0, 0)

# Read the entire file content
data = fo.read()
print("Updated file content:", data)

# Close the file
fo.close()
```

Following is the output of the above code −

```
Data read from position 10: rat
Updated file content: This is a cat race
```