**FILE HANDLING IN PYTHON**

**What is a File?**

A **file** is a **named location used for storing data**. For example, **main.py** is a file that is always used to store Python code. **File handling in Python** is a powerful and versatile tool that can be used to perform a wide range of operations.

## **Python File Handling**

Python supports file handling and **allows users to handle files** i.e., to read and write files, along with many other file handling options, to operate on files. Python treats files differently as text or binary and this is important. Each line of code includes a sequence of characters, and they form a text file. Each line of a file is terminated with a special character, called the **EOL** or **End of Line** characters like **comma** **{,}**or **newline character.** It ends the current line and tells the interpreter a new one has begun.

### **Advantages of File Handling in Python**

* **Versatility**: File handling in Python allows you to perform a **wide range of operations**, such as **creating, reading, writing, appending, renaming, and deleting files.**
* **Flexibility**: File handling in Python is highly flexible, as it allows you to **work with different file types (e.g. text files, binary files, CSV files, etc.)**, and to perform different operations on files (e.g. read, write, append, etc.).
* **User–friendly**: Python provides a **user-friendly interface** for file handling, making it **easy to create, read, and manipulate files**.
* **Cross-platform**: Python file-handling functions **work across different platforms** (e.g. Windows, Mac, Linux), allowing for **seamless integration and compatibility**.

### **Disadvantages of File Handling in Python**

* **Error-prone:** File handling operations in Python can be **prone to errors**, especially **if the** **code is not carefully written** or if there are **issues with the file system** (e.g. file permissions, file locks, etc.).
* **Security risks**: File handling in Python can also **pose security risks**, especially if the program accepts user input that can be used to **access or modify sensitive files on the system**.
* **Complexity**: File handling in Python can be **complex**, especially when working with more advanced file formats or operations. Careful attention must be paid to the code to **ensure that files are handled properly and securely**.
* **Performance**: **File handling operations in Python** can be **slower than other programming languages**, especially when **dealing with large files or performing complex operations**.

## **Python File Open**

Before performing any operation on the file like reading or writing, first, we have to open that file. For this, we should use **Python’s inbuilt function open()** but at the time of opening, we have to specify the mode, which represents the purpose of the opening file.

**f = open(filename, mode)**

Where the following mode is supported:

1. **r:**open an existing file for a read operation.
2. **w:** open an existing file for a write operation. If the file already contains some data, then it will be **overridden** but if the file is not present then it creates the file as well.
3. **a:**open an existing file for append operation. It **won’t override** existing data.
4. **r+:**  To read and write data into the file. The previous data in the file will be **overridden**.
5. **w+:** To write and read data. It will **override** existing data.
6. **a+:** To append and read data from the file. It **won’t override** existing data.
7. **x :** To create a new file,raises error if the file already exists.

## **Working in Read mode**

**Example 1:** The open command will open the Python file in the read mode and the for loop will print each line present in the file.

|  |
| --- |
| # a file named "geek", will be opened with the reading mode.  file **=** open('geek.txt', 'r')    # This will print every line one by one in the file  **for** each **in** file:      print (each) |

**Output:**

Hello world  
GeeksforGeeks  
123 456

**Example 2:** In this example, we will **extract a string that contains all characters** in the Python file then we can use **file.read()**.

|  |
| --- |
| # Python code to illustrate read() mode  file **=** open("geeks.txt", "r")  print (file.read()) |

**Output:**

Hello world  
GeeksforGeeks  
123 456

**Example 3:** In this example, we will see how we can read a file using the [**with**statement](https://www.geeksforgeeks.org/with-statement-in-python/) in Python.

|  |
| --- |
| # Python code to illustrate with()  with open("geeks.txt") as file:      data **=** file.read()    print(data) |

**Output:**

Hello world  
GeeksforGeeks  
123 456

**Example 4:** Another way to read a file is to **call a certain number of characters** like in the following code the interpreter will **read the** **first five characters of stored data and return it as a string**:

|  |
| --- |
| # Python code to illustrate read() mode character wise  file **=** open("geeks.txt", "r")  print (file.read(5)) |

**Output:**

Hello

## **Working in Write Mode**

**Example 1:** In this example, we will see how the write mode and the write() function is used to write in a file. The **close()** command **terminates all the resources in use and frees the system of this particular program**.

|  |
| --- |
| # Python code to create a file  file **=** open('geek.txt','w')  file.write("This is the write command")  file.write("It allows us to write in a particular file")  file.close() |

**Output:**

This is the write commandIt allows us to write in a particular file

**Example 2:** We can also use the written statement along with the  **with()** function.

|  |
| --- |
| # Python code to illustrate with() alongwith write()  with open("file.txt", "w") as f:      f.write("Hello World!!!") |

**Output:**

Hello World!!!

## **Working of Append Mode**

**Example:** For this example, we will use the Python file created in the previous example.

|  |
| --- |
| # Python code to illustrate append() mode  file **=** open('geek.txt', 'a')  file.write("This will add this line")  file.close() |

**Output:**

This is the write commandIt allows us to write in a particular fileThis will add this line

## **Implementing all the functions in File Handling**

|  |
| --- |
| **import** os    **def** create\_file(filename):  **try**:          with open(filename, 'w') as f:              f.write('Hello, world!\n')          print("File " **+** filename **+** " created successfully.")  **except** IOError:          print("Error: could not create file " **+** filename)    **def** read\_file(filename):  **try**:          with open(filename, 'r') as f:              contents **=** f.read()  **print**(contents)  **except** IOError:  **print**("Error: could not read file " **+** filename)    **def** append\_file(filename, text):  **try**:          with open(filename, 'a') as f:              f.write(text)  **print**("Text appended to file " **+** filename **+** " successfully.")  **except** IOError:          print("Error: could not append to file " **+** filename)    **def** rename\_file(filename, new\_filename):  **try**:          os.rename(filename, new\_filename)  **print**("File " **+** filename **+** " renamed to " **+** new\_filename **+** " successfully.")  **except** IOError:          print("Error: could not rename file " **+** filename)    **def** delete\_file(filename):  **try**:          os.remove(filename)  **print**("File " **+** filename **+** " deleted successfully.")  **except** IOError:  **print**("Error: could not delete file " **+** filename)      **if** \_\_name\_\_ **==** '\_\_main\_\_':      filename **=** "example.txt"      new\_filename **=** "new\_example.txt"        create\_file(filename)      read\_file(filename)      append\_file(filename, "This is some additional text.\n")      read\_file(filename)      rename\_file(filename, new\_filename)      read\_file(new\_filename)      delete\_file(new\_filename) |

**Output:**

File example.txt created successfully.  
Hello, world!  
Text appended to file example.txt successfully.  
Hello, world!  
This is some additional text.  
File example.txt renamed to new\_example.txt successfully.  
Hello, world!  
This is some additional text.  
File new\_example.txt deleted successfully.

Hello, world!  
This is some additional text.  
File new\_example.txt deleted successfully.

# BINARY FILE HANDLING

# Binary Files are stored in a binary format having digits 0’s and 1’s. For example, the number 9 in binary format is represented as ‘1001’. In this way, our computer stores each and every file in a machine-readable format in a sequence of binary digits. The structure and format of binary files depend on the type of file. Image files have different structures when compared to audio files. However, decoding binary files depends on the complexity of the file format.

# How do I read (or write) binary data in Python?

|  |  |
| --- | --- |
| **Mode** | **Description** |
| **rb** | Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode. |
| **rb+** | Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file. |
| **wb** | Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing. |
| **wb+** | Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing. |
| **ab** | Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing. |
| **ab+** | Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing. |

**Example:**

|  |
| --- |
| # Opening the binary file in binary mode as rb(read binary)  f **=** open("files.zip", mode**=**"rb")    # Reading file data with read() method  data **=** f.read()    # Printing our byte sequenced data  print(data)    # Closing the opened file  f.close() |

**Output:**

In the output, we see a sequence of byte data as bytes are the fundamental unit of binary representation.

*b’PK\x03\x04\x14\x00\x00\x00\x08\x00U\xbd\xebV\xc2=j\x87\x1e\x00\x00\x00!\x00\x00\x00\n\x00\x00\x00TODO11.txt\xe3\xe5JN,\xceH-/\xe6\xe5\x82\xc0\xcc\xbc\x92\xd4\x9c\x9c\xcc\x82\xc4\xc4\x12^.w7w\x00PK\x01\x02\x14\x00\x14\x00\x00\x00\x08\x00U\xbd\xebV\xc2=j\x87\x1e\x00\x00\x00!\x00\x00\x00\n\x00\x00\x00\x00\x00\x00\x00\x01\x00 \x00\x00\x00\x00\x00\x00\x00TODO11.txtPK\x05\x06\x00\x00\x00\x00\x01\x00\x01\x008\x00\x00\x00F\x00\x00\x00\x00\x00′*

**SERIALIZATION AND DESERIALIZATION**

**SERIALIZATION**

**Definition:** Serialization is the **process of converting a data structure or object into a format** (like a string, bytes, or a file) **that can be easily stored or transmitted and later reconstructed**.

**Purpose:** It allows the data to be saved to a file, sent over a network, or stored in a database, among other use cases.

**Example in Python**: The pickle module is commonly used for serializing Python objects. JSON and XML are other formats often used for serialization.

**DESERIALIZATION**

**Definition:** Deserialization is the **process of reconstructing a data structure or object from a serialized format back into its original form.**

**Purpose:** It allows the previously serialized data to be used within a program or application.

**Example in Python:** When using the **pickle** module, deserialization is achieved using the **load** or **loads** functions.

In Python, pickling and unpickling are terms associated with the **pickle** module, which is used for serializing and deserializing objects.

**PICKLING**

**Definition:** Pickling is the process of converting a Python object into a byte stream, which can be saved to a file or transmitted over a network.

**Usage:** The **pickle.dump()** function is typically used for pickling. It serializes the object and writes the byte stream to a file-like object.

**Example:**

import pickle

data = {'name': 'John', 'age': 25, 'city': 'Exampleville'}

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

pickle.dump(data, file)

**UNPICKLING**

**Definition:** Unpickling is the process of reconstructing a Python object from a byte stream, which was previously pickled.

**Usage:** The **pickle.load()** function is used for unpickling. It reads the byte stream from a file-like object and reconstructs the original object.

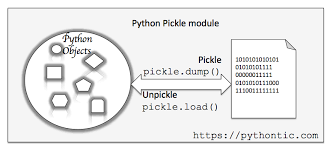
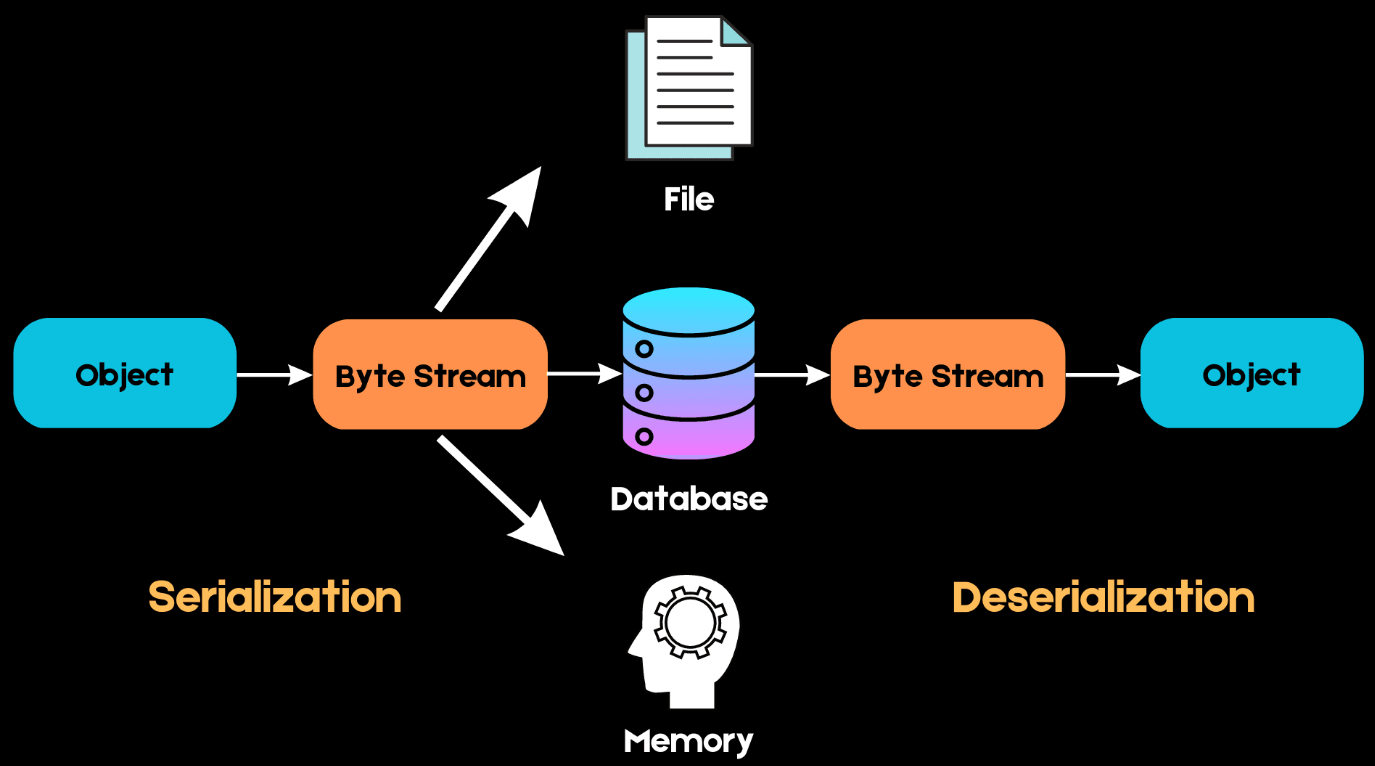
**Example:**

import pickle

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

loaded\_data = pickle.load(file)

print(loaded\_data)



**Load vs Loads**

1. **pickle.load(file)**:
   * **Definition:** This function is used to deserialize a pickled object from a file-like object.
   * **Usage:** You provide a file-like object (opened in binary mode **'rb'**) as an argument, and **pickle.load** reads the byte stream from that file, reconstructing the original Python object.

Example:

pythonCopy code

import pickle with open('data.pkl', 'rb') as file: loaded\_data = pickle.load(file)

1. **pickle.loads(bytes\_object)**:
   * **Definition:** This function is used to deserialize a pickled object from a bytes-like object.
   * **Usage:** Instead of reading from a file, **pickle.loads** takes a bytes-like object (containing the pickled data) as an argument and reconstructs the original Python object.

Example:

pythonCopy code

import pickle pickled\_data = b'\x80\x04\x95\x1e\x00\x00...' loaded\_data = pickle.loads(pickled\_data)

**Key Differences:**

* **pickle.load** reads from a file-like object, whereas **pickle.loads** takes a bytes-like object directly.
* Use **pickle.load** when you have pickled data stored in a file, and use **pickle.loads** when you have pickled data in-memory as bytes.
* **pickle.load** is commonly used with the **open** function to read from a file, while **pickle.loads** is used when the pickled data is already available as bytes.

In summary, the choice between **load** and **loads** depends on whether your pickled data is stored in a file or as bytes in memory.

**Dump vs Dumps**

1. **pickle.dump(obj, file)**:
   * **Definition:** This function is used to serialize and save a Python object to a file-like object.
   * **Usage:** You provide a file-like object (opened in binary mode **'wb'**) and the Python object to be pickled. **pickle.dump** writes the serialized byte stream to the specified file.

Example:

pythonCopy code

import pickle data = {'name': 'John', 'age': 25, 'city': 'Exampleville'} with open('data.pkl', 'wb') as file: pickle.dump(data, file)

1. **pickle.dumps(obj)**:
   * **Definition:** This function is used to serialize a Python object into a bytes-like object.
   * **Usage:** Instead of writing to a file, **pickle.dumps** takes the Python object as an argument and returns a bytes-like object containing the serialized data.

Example:

pythonCopy code

import pickle data = {'name': 'John', 'age': 25, 'city': 'Exampleville'} pickled\_data = pickle.dumps(data)

**Key Differences:**

* **pickle.dump** writes the serialized data to a file-like object, while **pickle.dumps** returns a bytes-like object containing the serialized data.
* Use **pickle.dump** when you want to save the pickled data to a file, and use **pickle.dumps** when you want to obtain the serialized data in-memory as bytes.
* **pickle.dump** is commonly used with the **open** function to write to a file, while **pickle.dumps** is used when you want to work with the serialized data directly in memory.

In summary, the choice between **dump** and **dumps** depends on whether you want to save the pickled data to a file or work with it as bytes in memory.

**WITH STATEMENT**

# What is the use of the WITH statement in Python?

* The **with statement** in Python **replaces a try-catch block with a simple shorthand**.
* More significantly, it **ensures that resources are closed immediately after processing**.
* A **context manager** is a **function or class that supports the with statement. A context manager enables you to open and close resources right when you want to**.
* The **open() function**, for example, **is a context manager**. When you **use the** **with statement to call the open() function**, the **file closes automatically** after you've processed it.
* To be considered a **context manager**, a class must implement the following **two methods** –
* **\_\_enter\_\_()**
* **\_\_exit\_\_()**
* When **with statement is called**, the **\_\_enter\_\_()** method is invoked.
* When you **exit the scope of the with block**, the **\_\_exit\_\_()** is invoked.
* With keyword is used not only to open a file in reading mode, but also to **assign an alias name to the opened file**.

**Example :** We can also **split lines while reading files in Python**. The **split()** function splits the variable when space is encountered. You can also split using any characters as you wish.

|  |
| --- |
| # Python code to illustrate split() function  with open("geeks.txt", "r") as file:      data **=** file.readlines()  **for** line **in** data:          word **=** line.split()          print (word) |

**Output:**

['Hello', 'world']  
['GeeksforGeeks']  
['123', '456']

## **Using “with” statement to open and read a file**

# input file path

inputFile = "ExampleTextFile.txt"

print("The lines of a given Text File are:")

# Opening the given file in read-only mode.

with open(inputFile, 'r') as fileData:

# Read the above file lines using readlines()

fileLines = fileData.readlines()

# Traverse in the each line of the text file

for textLine in fileLines:

# printing each line

print(textLine)

### **Output**

The lines of a given Text File are:

Good Morning this is Tutorials Point sample File

Consisting of Specific

Good source codes in Python,Seaborn,Scala

Summary and Explanation

**Stream Object in Python**

In Python, a **stream object** is a versatile tool for **handling a continuous flow of data, useful for reading or writing from different sources like files, memory, or networks**. It provides a step-by-step way to process data. A file object, a specific type of stream, is tailored for reading or writing files, offering methods like **read(), write(), and readline()**. So, a **file object is essentially a specialized kind of stream object crafted for file-related operations in Python**.

**File Objects in Python**

A file object **allows us to use, access and manipulate all the user accessible files**. One can read and write any such files. When a **file operation fails for an I/O-related reason**, the **exception IOError** is raised. This includes situations where the operation is not defined for some reason or writing a file opened for reading. Files have the following methods:

* **open():**Opens a file in given access mode.

**open(file\_address, access\_mode)**

Examples of accessing a file: A file can be opened with a built-in function called open(). This function takes in the file’s address and the access\_mode and returns a file object. There are different types of access\_modes:

**r:** Opens a file for reading only

**r+:** Opens a file for both reading and writing

**w:** Opens a file for writing only

**w+:** Open a file for writing and reading.

**a:** Opens a file for appending

**a+:** Opens a file for both appending and reading

When you add 'b' to the access modes you can read the file in binary format rather than the default text format. It is used when the file to be accessed is not in text.

* **read([size])**: It **reads the entire file and returns it contents in the form of a string**. Reads at most size bytes from the file (less if the read hits EOF before obtaining size bytes). If the size argument is negative or omitted, read all data until EOF is reached.

|  |
| --- |
| # Reading a file  f **=** open(\_\_file\_\_, 'r')    #read()  text **=** f.read(10)  print(text)  f.close() |

* **readline([size])**: It reads the first line of the file i.e till a newline character or an EOF in case of a file having a single line and returns a string. If the size argument is present and non-negative, it is a maximum byte count (including the trailing newline) and an incomplete line may be returned. An empty string is returned only when EOF is encountered immediately.

|  |
| --- |
| # Reading a line in a file  f **=** open(\_\_file\_\_, 'r')  #readline()  text **=** f.readline(20)  print(text)  f.close() |

* **readlines([sizehint])**: It reads the entire file line by line and updates each line to a list which is returned.Read until EOF using readline() and return a list containing the lines thus read. If the optional sizehint argument is present, instead of reading up to EOF, whole lines totalling approximately sizehint bytes (possibly after rounding up to an internal buffer size) are read.

|  |
| --- |
| # Reading a file  f **=** open(\_\_file\_\_, 'r')  #readline()  text **=** f.readlines(25)  print(text)  f.close() |

* **write(string)**: It writes the contents of string to the file. It has no return value. Due to buffering, the string may not actually show up in the file until the flush() or close() method is called.

|  |
| --- |
| # Writing a file  f **=** open(\_\_file\_\_, 'w')  line **=** 'Welcome Geeks\n'  #write()  f.write(line)  f.close() |

* **writelines(sequence)**: It is a sequence of strings to the file usually a list of strings or any other iterable data type. It has no return value.

|  |
| --- |
| # Writing a file  f **=** open(\_\_file\_\_, 'a+')  lines **=** f.readlines()  #writelines()  f.writelines(lines)  f.close() |

* **close()**: Used to close an open file. A closed file cannot be read or written any more.

|  |
| --- |
| # Opening and closing a file  f **=** open(\_\_file\_\_, 'r')  #close()  f.close() |

**STANDARD INPUT, OUTPUT AND ERROR MODES**

### Standard Input (stdin):

**Definition:** Standard input is the default input stream for a program. It allows the program to receive data from an external source, usually provided by the user via the keyboard. The primary function for reading standard input is the **input()** function.

**How it works:**

* The **input()** function is commonly used to read data from the standard input.
* When **input()** is called, the program waits for the user to enter data. Once the user presses Enter, the entered data is returned as a string.

There are a number of ways in which we can take input from stdin in Python.

* [sys](https://www.geeksforgeeks.org/python-sys-module/).stdin
* [input()](https://www.geeksforgeeks.org/python-input-function/)
* [fileinput.input()](https://www.geeksforgeeks.org/fileinput-input-in-python/)

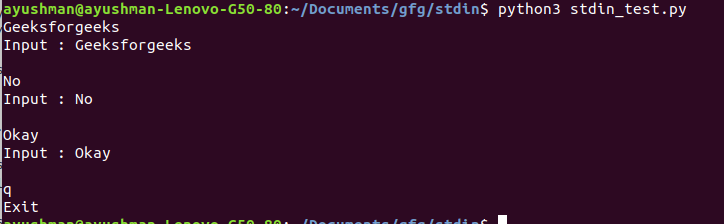
## Read Input From stdin in Python using sys.stdin

First we need to import sys module.**sys.stdin**can be used to get input from the command line directly. It used is for standard input. It internally calls the input() method. Furthermore, it, also, automatically adds ‘\n’ after each sentence.

### Example: Taking input using sys.stdin in a for-loop

|  |
| --- |
| **import** sys  **for** line **in** sys.stdin:  **if** 'q' **==** line.rstrip():  **break**      print(f'Input : {line}')  print("Exit") |

**Output**



## Read Input From stdin in Python using input()

The **input()** can be used to take input from the user while executing the program and also in the middle of the execution.

|  |
| --- |
| # this accepts the user's input  # and stores in inp  inp **=** input("Type anything")    # prints inp  print(inp) |

**Output:**



## Read Input From stdin in Python using fileinput.input()

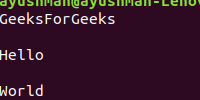
If we want to read more than one file at a time, we use **fileinput.input().** There are two ways to use fileinput.input(). To use this method, first, we need to import fileinput.

### **Example 1:** Reading multiple files by providing file names in fileinput.input() function argument

Here, we pass the name of the files as a tuple in the “files” argument. Then we loop over each file to read it. “sample.txt” and “no.txt” are two files present in the same directory as the Python file.

|  |
| --- |
| **import** fileinput  with fileinput.input(files**=**('sample.txt', 'no.txt')) as f:  **for** line **in** f:          print(line) |

**Output:**



### **Example 2:** Reading multiple files by passing file names from command line using fileinput module

Here, we pass the file name as a positional argument in the command line. fileargument parses the argument and reads the file and displays the content of the file.

|  |
| --- |
| **import** fileinput  **for** f **in** fileinput.input():      print(f) |

### Errors :

In Python, errors are typically categorized into three main types: **syntax errors, runtime errors, and logical errors**. Each of these error types has different causes and can be addressed in various ways. Here's a brief overview of each:

1. **SyntaxError:**
   * **Cause:** This exception is raised when the interpreter encounters a syntax error in the code, such as a misspelled keyword, a missing colon, or an unbalanced parenthesis.
   * **Example:**

print("Hello, World!"

1. **IndentationError:**
   * **Cause:** Incorrect indentation.
   * **Example:**

def my\_function():

print("Indented incorrectly")

1. **NameError:**
   * **Cause:** This exception is raised when a variable or function name is not found in the current scope.
   * **Example:**

print(undefined\_variable)

1. **TypeError:**
   * **Cause:** This exception is raised when an operation or function is applied to an object of the wrong type, such as adding a string to an integer.
   * **Example:**

result = "10" + 5

1. **ValueError:**
   * **Cause:** This exception is raised when a function or method is called with an invalid argument or input, such as trying to convert a string to an integer when the string does not represent a valid integer.
   * **Example:**

number = int("abc")

1. **ZeroDivisionError:**
   * **Cause:** This exception is raised when an attempt is made to divide a number by zero.
   * **Example:**

result = 10 / 0

1. **IndexError:**
   * **Cause:** This exception is raised when an index is out of range for a list, tuple, or other sequence types.
   * **Example:**

my\_list = [1, 2, 3]

print(my\_list[5])

1. **FileNotFoundError:**
   * **Cause:** Trying to open or access a file that doesn't exist.
   * **Example:**

with open("nonexistent\_file.txt", "r") as file:

content = file.read()

1. **AttributeError:**
   * **Cause:** This exception is raised when an attribute or method is not found on an object, such as trying to access a non-existent attribute of a class instance.
   * **Example:**

**my\_string = "Hello, World!"**

**length = my\_string.len()**

1. **AssertionError:**
   * **Cause:** An **assert** statement fails.
   * **Example:**

assert 2 + 2 == 5, "Math is not working as expected"

1. **KeyError:** This exception is raised when a key is not found in a dictionary.
2. **IOError:** This exception is raised when an I/O operation, such as reading or writing a file, fails due to an input/output error.
3. **ImportError:** This exception is raised when an import statement fails to find or load a module.
4. **EOFError:** When reading from standard input, an EOFError may occur if the end-of-file (EOF) is reached unexpectedly.This can happen if input is being read from a file and there is no more content to read.

Here's a brief explanation of standard input in Python:

1. **input() Function:**
   * The **input()** function is used to read a line of text from the standard input (usually the keyboard).
   * It takes a prompt as an optional argument, which is displayed to the user before waiting for input.
   * The input is read as a string and can be stored in a variable for further processing.

pythonCopy code

user\_input = input("Enter your name: ") print(f"Hello, {user\_input}!")

1. **Reading Numeric Input:**
   * If you need to read numeric input (integer or float), you can convert the input using **int()** or **float()** functions.

pythonCopy code

age = int(input("Enter your age: "))

1. **Standard Input Redirection:**
   * Python scripts can also read input from other sources by using standard input redirection.
   * This allows you to read input from a file or another program instead of directly from the keyboard.

pythonCopy code

# Assuming input is redirected from a file with open('input.txt', 'r') as file: user\_input = file.readline()

1. **EOFError:**
   * When reading from standard input, an **EOFError** may occur if the end-of-file (EOF) is reached unexpectedly.
   * This can happen if input is being read from a file and there is no more content to read.
2. **Example: Simple Input Handling:**
   * Here's a simple example of using **input()** to get user input and perform some processing:

pythonCopy code

# Get user input name = input("Enter your name: ") # Process the input if name: print(f"Hello, {name}!") else: print("You didn't enter a name.")

1. **Raw Input (Python 2 vs. Python 3):**
   * In Python 2, the **raw\_input()** function was used for reading strings from the user, while **input()** evaluated user input as Python code. In Python 3, **raw\_input()** has been removed, and **input()** is used for both purposes.

Standard input is a fundamental concept in interactive programming, allowing Python programs to interact with users by receiving input during runtime. It's commonly used for building interactive command-line applications and for general user interaction in various scenarios.

Certainly! In Python, standard input (often abbreviated as stdin) and standard output (often abbreviated as stdout) are streams that facilitate communication between a program and the external environment, typically the terminal or console where the program is executed.

**Example:**

pythonCopy code

name = input("Enter your name: ") print("Hello, " + name + "!")

In this example, the **input()** function is used to read the user's name from the standard input, and the entered name is then used in the output.

### Standard Output (stdout):

**Definition:** Standard output is the default output stream for a program. It allows the program to display information or results to the user on the terminal or console.

**How it works:**

* The **print()** function is commonly used to write data to the standard output.
* Data passed to **print()** is displayed on the console.

This is a built-in Python module that contains parameters specific to the system i.e. it contains variables and methods that interact with the interpreter and are also governed by it.

## **sys.stdout**

A built-in file object that is analogous to the interpreter’s standard output stream in Python. stdout is used to display output directly to the screen console. Output can be of any form, it can be output from a print statement, an expression statement, and even a prompt direct for input. By default, streams are in text mode. In fact, wherever a print function is called within the code, it is first written to sys.stdout and then finally on to the screen.

sys.stdout.write() serves the same purpose as the object stands for except it prints the number of letters within the text too when used in interactive mode. Unlike print, sys.stdout.write doesn’t switch to a new line after one text is displayed. To achieve this one can employ a new line escape character(\n).

**Syntax:**

sys.stdout.write(<some string text here>)

**Example 1:**

|  |
| --- |
| **import** sys  sys.stdout.write('gfg') |

**Output**

gfg

**Example 2:**

|  |
| --- |
| # script mode  **import** sys  sys.stdout.write('gfg')  sys.stdout.write('geeks')  sys.stdout.write('\n')  sys.stdout.write('for geeks') |

**Output**

gfggeeks

for geeks

stdout can be also be used to print multiple elements. Not just this stdout can be assigned to another variable as long as it supports write().

**Example 3:**

## Python3

|  |
| --- |
| **import** sys  # stdout assigned to a variable  var **=** sys.stdout  arr **=** ['geeks', 'for', 'geeks']  # printing everything in the same line  **for** i **in** arr:      var.write(i)  # printing everything in a new line  **for** j **in** arr:      var.write('\n'**+**j) |

**Output:**

geeksforgeeks

geeks

for

geeks