

Read Files in Python

Read files in python - Intro

One of the most common tasks that you can do with Python is reading and writing files.

Whether it's writing to a simple text file, reading a complicated server log, or even analyzing raw byte data, all of these situations require reading or writing a file.



What is a file?

Before we can go into how to work with files in Python, it's important to understand what exactly a file is and how modern operating systems handle some of their aspects.

A file is a contiguous set of bytes used to store data.

This data is organized in a specific format and can be anything as simple as a text file or as complicated as a program executable.

In the end, these byte files are then translated into binary 1 and 0 for easier processing by the computer.

Files on most modern file systems are composed of three main parts:

1. **Header:** metadata about the contents of the file (file name, size, type, and so on)
2. **Data:** contents of the file as written by the creator or editor
3. **End of file (EOF):** special character that indicates the end of the file



Opening and Closing a File in Python

When you want to work with a file, the first thing to do is to open it.

This is done by invoking the **open()** built-in function.

- **open()** has a single required argument that is the path to the file.
- **open()** has a single return, the file object:

```
file = open('dog_breeds.txt')
```

After you open a file, the next thing to learn is how to close it.

Opening and Closing a File in Python

It's important to remember that it's your responsibility to close the file.

In most cases, upon termination of an application or script, a file **will be closed eventually**.

There is no guarantee when exactly that will happen.

This can lead to unwanted behavior including resource leaks. It's also a best practice within Python (Pythonic) to make sure that your code behaves in a way that is well defined and reduces any unwanted behavior.

When you're manipulating a file, there are two ways that you can use to ensure that a file is closed properly, even when encountering an error.

Opening and Closing a File in Python - Close a file

The first way to close a file is to use the `try-finally` block:

```
reader = open('dog_breeds.txt')
try:
    # Further file processing goes here
finally:
    reader.close()
```

Opening and Closing a File in Python - Close a file

The first way to close a file is to use the `try-finally` block:

```
reader = open('dog_breeds.txt')
try:
    # Further file processing goes here
finally:
    reader.close()
```

You can use also the **with** statement

```
with open('dog_breeds.txt') as reader:
    # Further file processing goes here
```

Opening and Closing a File in Python - Close a file

The first way to close a file is to use the `try-finally` block:

```
reader = open('dog_breeds.txt')
try:
    # Further file processing goes here
finally:
    reader.close()
```

You can use also the **with** statement

```
with open('dog_breeds.txt') as reader:
    # Further file processing goes here
```

The **with** statement automatically takes care of closing the file once it leaves the **with** block, even in cases of error.

I highly recommend that you use the **with** statement as much as possible, as it allows for cleaner code and makes handling any unexpected errors easier for you.

The with statement

One common problem you'll face in programming is how to properly manage external resources, such as **files**, **locks**, and **network connections**.

Sometimes, a program will **retain those resources forever**, even if you no longer need them.

This kind of issue is called a memory leak because the available memory gets reduced every time you create and open a new instance of a given resource without closing an existing one.



The with statement

Managing resources properly is often a tricky problem.

For example you should perform some cleanup actions, such as:

- **closing a file**
- **releasing a lock**
- **closing a network connection**

If you forget to perform these cleanup actions, then your application keeps the resource alive.

This might compromise valuable system resources, such as memory and network bandwidth.

The with statement - Example 1

A common problem that can arise when developers are working with databases is when a program keeps creating new connections **without releasing or reusing them**.

In that case, the database **back end can stop accepting new connections**.

This might require an admin to log in and manually kill those stale connections to make the database usable again.



The with statement - Example 2

Another frequent issue shows up when developers are working with files.

Writing text to files is usually a buffered operation.

This means that calling `.write()` on a file won't immediately result in writing text to the physical file but to a temporary buffer.

When the buffer isn't full and developers forget to call `.close()`, part of the data can be lost forever.

The with statement - Context Manager

The purpose of the **with** statement is to ensure that resources are properly managed and cleaned up, regardless of how the block is exited.

This is particularly useful for handling resources such as files, network connections, and locks.

On the right you can see a code snippet containing a custom context manager called **MyResource**

```
class MyResource:
    def __enter__(self):
        # Code to acquire resource or setup
        print("Resource acquired")
        # Return the resource itself (optional)
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        # Code to release resource or cleanup
        print("Resource released")
        # Handle exceptions (optional)
        if exc_type is not None:
            print(f"Exception type: {exc_type}")
            print(f"Exception value: {exc_value}")
            print(f"Traceback: {traceback}")
        # Return False to propagate the exception, True to suppress it
        return False

# Using the custom context manager with the 'with' statement
with MyResource() as resource:
    print("Inside the with block")
    # Simulate some operation that might cause an exception
    # Uncomment the following line to test exception handling
    # raise ValueError("An error occurred")
```

The with statement - Context Manager



The context manager MyResource contains **two functions**:

- `__enter__`:
 - This method is called when the execution flow **enters** the with block.
 - It prints "Resource acquired" to indicate that the resource has been acquired.
 - It can return the resource itself (or any other value) that will be assigned to the variable after the `as` keyword in the with statement.

```
class MyResource:
    def __enter__(self):
        # Code to acquire resource or setup
        print("Resource acquired")
        # Return the resource itself (optional)
        return self

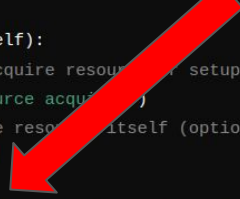
    def __exit__(self, exc_type, exc_value, traceback):
        # Code to release resource or cleanup
        print("Resource released")
        # Handle exceptions (optional)
        if exc_type is not None:
            print(f"Exception type: {exc_type}")
            print(f"Exception value: {exc_value}")
            print(f"Traceback: {traceback}")
        # Return False to propagate the exception, True to suppress it
        return False

# Using the custom context manager with the 'with' statement
with MyResource() as resource:
    print("Inside the with block")
    # Simulate some operation that might cause an exception
    # Uncomment the following line to test exception handling
    # raise ValueError("An error occurred")
```

The with statement - Context Manager

The context manager MyResource contains **two functions**:

- `__exit__`:
 - This method is called when the execution flow **leaves** the with block, regardless of whether an exception was raised or not.
 - It prints "Resource released" to indicate that the resource has been released.
 - If an exception occurs within the with block, **exc_type**, **exc_value**, and **traceback** will contain information about the exception.
 - The method can handle the exception (e.g., logging it) and must return False to propagate the exception, or True to suppress it.



```
class MyResource:
    def __enter__(self):
        # Code to acquire resource and setup
        print("Resource acquired")
        # Return the resource itself (optional)
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        # Code to release resource or cleanup
        print("Resource released")
        # Handle exceptions (optional)
        if exc_type is not None:
            print(f"Exception type: {exc_type}")
            print(f"Exception value: {exc_value}")
            print(f"Traceback: {traceback}")
        # Return False to propagate the exception, True to suppress it
        return False

# Using the custom context manager with the 'with' statement
with MyResource() as resource:
    print("Inside the with block")
    # Simulate some operation that might cause an exception
    # Uncomment the following line to test exception handling
    # raise ValueError("An error occurred")
```

The with statement - Context Manager

Using the **with** Statement:

1. The with statement is used to execute the block of code with the context manager.
2. The MyResource object is instantiated and the `__enter__` method is called.
3. The block of code inside the with statement is executed.
4. The `__exit__` method is called after the block is executed, even if an exception occurs.
5. If an exception occurs, it is propagated unless `__exit__` returns True.

```
class MyResource:
    def __enter__(self):
        # Code to acquire resource or setup
        print("Resource acquired")
        # Return the resource itself (optional)
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        # Code to release resource or cleanup
        print("Resource released")
        # Handle exceptions (optional)
        if exc_type is not None:
            print(f"Exception type: {exc_type}")
            print(f"Exception value: {exc_value}")
            print(f"Traceback: {traceback}")
        # Return False to propagate the exception, True to suppress it
        return False

# Using the custom context manager with the 'with' statement
with MyResource() as resource:
    print("Inside the with block")
    # Simulate some operation that might cause an exception
    # Uncomment the following line to test exception handling
    # raise ValueError("An error occurred")
```


The with statement - Context Manager

Exception Handling:

Uncommenting the line `raise ValueError("An error occurred")` within the with block will trigger an exception.

The `__exit__` method will handle the exception, print the exception details, and return `False` to propagate the exception.

```
class MyResource:
    def __enter__(self):
        # Code to acquire resource or setup
        print("Resource acquired")
        # Return the resource itself (optional)
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        # Code to release resource or cleanup
        print("Resource released")
        # Handle exceptions (optional)
        if exc_type is not None:
            print(f"Exception type: {exc_type}")
            print(f"Exception value: {exc_value}")
            print(f"Traceback: {traceback}")
        # Return False to propagate the exception, True to suppress it
        return False

# Using the custom context manager with the 'with' statement
with MyResource() as resource:
    print("Inside the with block")
    # Simulate some operation that might cause an exception
    # Uncomment the following line to test exception handling
    # raise ValueError("An error occurred")
```

The with statement - A Non Naive Example!

Non Naive Real Example:

Let's suppose you want to make some query on a remote database.

You can create a class called **DatabaseConnection** that has the responsibility to connect and disconnect from the remote and to execute a query!

A pretty standard way to operate!

```
class DatabaseConnection:
    def __init__(self, db_name):
        self.db_name = db_name
        self.connection = None

    def connect(self):
        print(f"Connecting to database: {self.db_name}")
        # Simulate a database connection
        self.connection = f"Connection to {self.db_name}"

    def disconnect(self):
        print(f"Disconnecting from database: {self.db_name}")
        self.connection = None

    def commit(self):
        print("Committing transaction")

    def rollback(self):
        print("Rolling back transaction")

    def execute_query(self, query):
        print(f"Executing query: {query}")
        # Simulate a query execution
        return f"Results of '{query}'"
```

The with statement - A Non Naive Example!

Non Naive Real Example:

You can create a **DatabaseContextManager** that helps you managing every aspect of the process.

```
class DatabaseContextManager:
    def __init__(self, db_name):
        self.db = DatabaseConnection(db_name)

    def __enter__(self):
        self.db.connect()
        return self.db

    def __exit__(self, exc_type, exc_value, traceback):
        if exc_type is not None:
            # If an exception occurred, roll back the transaction
            self.db.rollback()
            print(f"Exception type: {exc_type}")
            print(f"Exception value: {exc_value}")
            print(f"Traceback: {traceback}")
        else:
            # Commit the transaction if no exception occurred
            self.db.commit()

        # Always disconnect from the database
        self.db.disconnect()

        # Return False to propagate the exception, True to suppress
        return False
```

The with statement - A Non Naive Example!

Non Naive Real Example:

You can create a **DatabaseContextManager** that helps you managing every aspect of the process.

`__init__` declare a new **DatabaseConnection** object that you can use later in the `__enter__` function.

```
class DatabaseContextManager:
    def __init__(self, db_name):
        self.db = DatabaseConnection(db_name)

    def __enter__(self):
        self.db.connect()
        return self.db

    def __exit__(self, exc_type, exc_value, traceback):
        if exc_type is not None:
            # If an exception occurred, roll back the transaction
            self.db.rollback()
            print(f"Exception type: {exc_type}")
            print(f"Exception value: {exc_value}")
            print(f"Traceback: {traceback}")
        else:
            # Commit the transaction if no exception occurred
            self.db.commit()

        # Always disconnect from the database
        self.db.disconnect()

        # Return False to propagate the exception, True to suppress
        return False
```

The with statement - A Non Naive Example!

Non Naive Real Example:

In the `__enter__` function you can connect to the remote database and return the `DatabaseConnection` object

```
class DatabaseContextManager:
    def __init__(self, db_name):
        self.db = DatabaseConnection(db_name)

    def __enter__(self):
        self.db.connect()
        return self.db

    def __exit__(self, exc_type, exc_value, traceback):
        if exc_type is not None:
            # If an exception occurred, roll back the transaction
            self.db.rollback()
            print(f"Exception type: {exc_type}")
            print(f"Exception value: {exc_value}")
            print(f"Traceback: {traceback}")
        else:
            # Commit the transaction if no exception occurred
            self.db.commit()

        # Always disconnect from the database
        self.db.disconnect()

        # Return False to propagate the exception, True to suppress
        return False
```

The with statement - A Non Naive Example!

Non Naive Real Example:

Finally the `__exit__` function look for exceptions.

If there are no exceptions it can commit the changes, otherwise it does the rollback.

Before exiting the `DatabaseConnection` object call the `disconnect` function to close the connection and then returns `False` propagating eventual exceptions occurred!

```
class DatabaseContextManager:
    def __init__(self, db_name):
        self.db = DatabaseConnection(db_name)

    def __enter__(self):
        self.db.connect()
        return self.db

    def __exit__(self, exc_type, exc_value, traceback):
        if exc_type is not None:
            # If an exception occurred, roll back the transaction
            self.db.rollback()
            print(f"Exception type: {exc_type}")
            print(f"Exception value: {exc_value}")
            print(f"Traceback: {traceback}")
        else:
            # Commit the transaction if no exception occurred
            self.db.commit()

        # Always disconnect from the database
        self.db.disconnect()

        # Return False to propagate the exception, True to suppress
        return False
```

The with statement - A Non Naive Example!

Non Naive Real Example:

How can i use it?

Easy!


```
def main():  
    db_name = 'example.db'  
    query = 'SELECT * FROM users'  
  
    # Use the custom context manager to handle the database connection  
    try:  
        with DatabaseContextManager(db_name) as db:  
            results = db.execute_query(query)  
            print(f"Query results: {results}")  
            # Uncomment the next line to simulate an exception  
            # raise ValueError("Simulated error during database operation")  
    except Exception as e:  
        print(f"Caught an exception: {e}")
```

Opening and Closing a File in Python - Modes

Most likely, you'll also want to use the second positional argument, `mode`.

This argument is a string that contains multiple characters to represent how you want to open the file.

The default and most common is `'r'`, which represents opening the file in read-only mode as a text file:



```
with open('dog_breeds.txt', 'r') as reader:  
    # Further file processing goes here
```

Character	Meaning
'r'	Open for reading (default)
'w'	Open for writing, truncating (overwriting) the file first
'rb' or 'wb'	Open in binary mode (read/write using byte data)

Opening and Closing a File in Python - Modes

Most likely, you'll also want to use the second positional argument, `mode`.

This argument is a string that contains multiple characters to represent how you want to open the file.

The default and most common is `'r'`, which represents opening the file in read-only mode as a text file:

```
with open('dog_breeds.txt', 'r') as reader:  
    # Further file processing goes here
```

Character	Meaning
'r'	Open for reading (default)
'w'	Open for writing, truncating (overwriting) the file first
'rb' or 'wb'	Open in binary mode (read/write using byte data)

Opening and Closing a File in Python - Modes

Most likely, you'll also want to use the second positional argument, `mode`.

This argument is a string that contains multiple characters to represent how you want to open the file.

The default and most common is `'r'`, which represents opening the file in read-only mode as a text file:

```
with open('dog_breeds.txt', 'r') as reader:  
    # Further file processing goes here
```

Character	Meaning
'r'	Open for reading (default)
'w'	Open for writing, truncating (overwriting) the file first
'rb' or 'wb'	Open in binary mode (read/write using byte data)

Opening and Closing a File in Python - Modes

Most likely, you'll also want to use the second positional argument, `mode`.

This argument is a string that contains multiple characters to represent how you want to open the file.

The default and most common is `'r'`, which represents opening the file in read-only mode as a text file:

```
with open('dog_breeds.txt', 'r') as reader:  
    # Further file processing goes here
```

Character	Meaning
'r'	Open for reading (default)
'w'	Open for writing, truncating (overwriting) the file first
'rb' or 'wb'	Open in binary mode (read/write using byte data)

Reading and Writing Opened Files

```
>>> with open('dog_breeds.txt', 'r') as reader:
>>>     # Read & print the entire file
>>>     print(reader.read())
Pug
Jack Russell Terrier
English Springer Spaniel
German Shepherd
Staffordshire Bull Terrier
Cavalier King Charles Spaniel
Golden Retriever
West Highland White Terrier
Boxer
Border Terrier
```

Method	What It Does
<code>.read(size=-1)</code>	This reads from the file based on the number of size bytes. If no argument is passed or None or -1 is passed, then the entire file is read.
<code>.readline(size=-1)</code>	This reads at most size number of characters from the line. This continues to the end of the line and then wraps back around. If no argument is passed or None or -1 is passed, then the entire line (or rest of the line) is read.
<code>.readlines()</code>	This reads the remaining lines from the file object and returns them as a list.

Reading and Writing Opened Files

```
>>> with open('dog_breeds.txt', 'r') as reader:
>>>     # Read & print the first 5 characters of the line 5 times
>>>     print(reader.readline(5))
>>>     # Notice that line is greater than the 5 chars and continues
>>>     # down the line, reading 5 chars each time until the end of the
>>>     # line and then "wraps" around
>>>     print(reader.readline(5))
>>>     print(reader.readline(5))
>>>     print(reader.readline(5))
>>>     print(reader.readline(5))
Pug
```

```
Jack
Russe
ll Te
rrier
```

Method

What It Does

`.read(size=-1)`

This reads from the file based on the number of size bytes. If no argument is passed or None or -1 is passed, then the entire file is read.


`.readline(size=-1)`

This reads at most size number of characters from the line. This continues to the end of the line and then wraps back around. If no argument is passed or None or -1 is passed, then the entire line (or rest of the line) is read.

`.readlines()`

This reads the remaining lines from the file object and returns them as a list.

Reading and Writing Opened Files



```
>>> f = open('dog_breeds.txt')
>>> f.readlines() Returns a list object
['Pug\n', 'Jack Russell Terrier\n', 'English Springer Spaniel\n', 'German Shep
```

Equivalent methods

```
>>> f = open('dog_breeds.txt')
>>> list(f)
['Pug\n', 'Jack Russell Terrier\n', 'English Springer Spaniel\n', 'German Shep
```



Method	What It Does
<code>.read(size=-1)</code>	This reads from the file based on the number of size bytes. If no argument is passed or None or -1 is passed, then the entire file is read.
<code>.readline(size=-1)</code>	This reads at most size number of characters from the line. This continues to the end of the line and then wraps back around. If no argument is passed or None or -1 is passed, then the entire line (or rest of the line) is read.
<code>.readlines()</code>	This reads the remaining lines from the file object and returns them as a list.

Iterating Over Each Line in the File - readline()

readline read a single line at time until it encounters the EOF token

```
>>> with open('dog_breeds.txt', 'r') as reader:
>>>     # Read and print the entire file line by line
>>>     line = reader.readline()
>>>     while line != '': # The EOF char is an empty string
>>>         print(line, end='')
>>>         line = reader.readline()
Pug
Jack Russell Terrier
English Springer Spaniel
German Shepherd
Staffordshire Bull Terrier
Cavalier King Charles Spaniel
Golden Retriever
West Highland White Terrier
Boxer
Border Terrier
```

Iterating Over Each Line in the File - readlines()

readlines read all the lines within the file and put them into a list and then we access them using a for loop

```
>>> with open('dog_breeds.txt', 'r') as reader:  
>>>     for line in reader.readlines():  
>>>         print(line, end='')
```

```
Pug  
Jack Russell Terrier  
English Springer Spaniel  
German Shepherd  
Staffordshire Bull Terrier  
Cavalier King Charles Spaniel  
Golden Retriever  
West Highland White Terrier  
Boxer  
Border Terrier
```


Iterating Over Each Line in the File - readlines()

Iterating over the file object itself is the best way to read from a file.

It is more Pythonic and can be quicker and more memory efficient.

Therefore, it is suggested you use this instead.

```
>>> with open('dog_breeds.txt', 'r') as reader:
>>>     # Read and print the entire file line by line
>>>     for line in reader:
>>>         print(line, end='')
Pug
Jack Russell Terrier
English Springer Spaniel
German Shepherd
Staffordshire Bull Terrier
Cavalier King Charles Spaniel
Golden Retriever
West Highland White Terrier
Boxer
Border Terrier
```

Write on a file

Now let's dive into writing files.

As with reading files, file objects have multiple methods that are useful for writing to a file

write() can write a string into a file

writelines() instead can write a list of strings into a file

Remember, you should use the `\n` char to get a new line!

Method	What It Does
<code>.write(string)</code>	This writes the string to the file.
<code>.writelines(seq)</code>	This writes the sequence to the file. No line endings are appended to each sequence item. It's up to you to add the appropriate line ending(s).

Write on a file

Now let's dive into writing files.

As with reading files, file objects have multiple methods that are useful for writing to a file

write() can write a string into a file



writelines() instead can write a list of strings into a file

Remember, you should use the \n char to get a new line!

```
# Open a file in write mode
with open('example_write.txt', 'w') as file:
    # Write a single string to the file
    file.write("This is the first line.\n")
    file.write("This is the second line.\n")
    file.write("This is the third line.\n")
```

Write on a file

Now let's dive into writing files.

As with reading files, file objects have multiple methods that are useful for writing to a file

write() can write a string into a file

writelines() instead can write a list of strings into a file

Remember, you should use the \n char to get a new line!

```
# Open a file in write mode
with open('example_writelines.txt', 'w') as file:
    # Write a list of strings to the file
    lines = [
        "This is the first line.\n",
        "This is the second line.\n",
        "This is the third line.\n"
    ]
    file.writelines(lines)
```

