

File handling and Reading/ Writing Files

File handling in Python is a powerful and versatile tool that can be used to perform a wide range of operations. However, it is important to carefully consider the advantages and disadvantages of file handling when writing Python programs, to ensure that the code is secure, reliable, and performs well.

Python File Handling

Python too 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. The concept of file handling has stretched over various other languages, but the implementation is either complicated or lengthy, but like other concepts of Python, this concept here is also easy and short. 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. Let's start with the reading and writing files.

Working of open() Function in Python

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:

r: open an existing file for a read operation.

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.

a: open an existing file for append operation. It won't override existing data.

r+: To read and write data into the file. The previous data in the file will be overridden.

w+: To write and read data. It will override existing data.

a+: To append and read data from the file. It won't override existing data.

```
# a file named "test", will be opened with the
reading mode.
file = open('test.txt', 'r')

# This will print every line one by one in the file
for each in file:
    print(each)
```



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

```
# Python code to illustrate read() mode
file = open("test.txt", "r")
print (file.read())
```

Example 3: In this example, we will see how we can read a file using the `with` statement

```
# Python code to illustrate with()
with open("test.txt") as file:
    data = file.read()

print(data)
```

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("test.txt", "r")
print(file.read(5))
```

Example 5:

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("test.txt", "r") as file:
    data = file.readlines()
    for line in data:
        word = line.split()
        print(word)
```

Creating a File using the `write()` Function

Just like reading a file in Python, there are a number of ways to write in a file in Python. Let us see how we can write the content of a file using the `write()` function in Python.



Working in Write Mode

Let's see how to create a file and how the write mode works.

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('test.txt', 'w')
file.write("This is the write command")
file.write("It allows us to write in a particular
file")
file.close()
```

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("RPA Mastery Bootcamp")
```

Working of Append Mode

Let us see how the append mode works.

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

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

Delete Files

In Python, there are multiple methods to delete files. Each method offers different functionalities and advantages. Below are some of the common methods to delete files in Python:

Using os.remove():

This method is part of the os module in Python and is used to delete a file by providing the file path as an argument. It raises an exception if the file does not exist or if there are permission issues.



```
import os

file_path = 'path/to/your/file.txt'

os.remove(file_path)
```

Using `os.unlink()`:

Similar to `os.remove()`, `os.unlink()` is used to delete a file. The difference is that `os.unlink()` is an alias for `os.remove()` and works the same way.

```
import os

file_path = 'path/to/your/file.txt'

os.remove(file_path)
```

Using `os.rmdir()`:

This method is used to remove an empty directory. It raises an exception if the directory is not empty or if there are permission issues.

```
import os

directory_path = 'path/to/your/directory'

os.rmdir(directory_path)
```

Using `shutil.rmtree()`:

This method, part of the `shutil` module, is used to remove a directory and all its contents (including subdirectories and files). Exercise caution when using this method, as it can permanently delete a directory and its contents.

```
import shutil

directory_path = 'path/to/your/directory'

shutil.rmtree(directory_path)
print("Directory and its contents deleted successfully.")
```

Create Directory

In Python, you can create a directory (folder) using different methods. Here are some common methods to create a directory:



Using `os.mkdir()`:

This method is part of the `os` module and is used to create a directory with the specified name. It raises an exception if the directory already exists or if there are permission issues.

```
import os
directory_name = 'new_directory'
os.mkdir(directory_name)
print("Directory '{}' created successfully.".format(directory_name))
```

Using `os.makedirs()`:

The `os.makedirs()` method, also part of the `os` module, is used to create a directory along with its parent directories (if they do not exist). This is helpful when you need to create nested directories.

```
import os

nested_directory =
'parent_directory/child_directory'

os.makedirs(nested_directory)
print("Nested directory '{}' created successfully.".format(nested_directory))
```

Using `os.makedirs()` with `exist_ok=True`:

To create a directory using `os.makedirs()` and avoid an exception if the directory already exists, you can use the `exist_ok=True` argument.

```
import os

directory_name = 'new_directory'

os.makedirs(directory_name, exist_ok=True)
print("Directory '{}' created successfully or already exists.".format(directory_name))
```

Get all files from directories



To get all files from directories in Python, you can use various methods and modules. Here are some common methods:

Using `os.listdir()`:

The `os.listdir()` method from the `os` module lists all the files and directories in a given directory. You can then filter the results to obtain only the files.

```
import os

directory_path = 'path/to/your/directory'

# Get all files and directories in the specified directory
files_and_dirs = os.listdir(directory_path)

# Filter only the files from the list
files_list = [file for file in files_and_dirs if
os.path.isfile(os.path.join(directory_path, file))]

print(files_list)
```

Using `os.walk()`:

The `os.walk()` method is an efficient way to recursively traverse through a directory and its subdirectories, providing access to all files and directories within them.

```
import os

directory_path = 'path/to/your/directory'

files_list = []
for root, dirs, files in os.walk(directory_path):
    for file in files:
        files_list.append(os.path.join(root, file))

print(files_list)
```

Using `glob.glob()`:

The `glob.glob()` method from the `glob` module allows you to search for files using wildcard patterns.



```
import glob

directory_path = 'path/to/your/directory'

# Search for all files in the specified directory
using wildcard pattern
files_list = glob.glob(os.path.join(directory_path,
'*'))

print(files_list)
```

Using `pathlib.Path.glob()`:

If you prefer the object-oriented approach provided by the `pathlib` module, you can use the `glob()` method of `Path` objects.

```
from pathlib import Path

directory_path = Path('path/to/your/directory')

# Search for all files in the specified directory
using wildcard pattern
files_list = list(directory_path.glob('*'))

print(files_list)
```

Get current date

Library used to get the date

Import datetime

```
import datetime

current_date = datetime.date.today()

print(current_date)
```

The `datetime.date.today()` function returns the current date as a `datetime.date` object, which represents the date without the time component.



When you print `current_date`, it will be displayed in the format `YYYY-MM-DD`, where `YYYY` is the year, `MM` is the month, and `DD` is the day.

To get the current date and time in Python, you can use the `datetime` module. The `datetime` module provides the `datetime` class, which combines both date and time information. Here's how you can get the current date and time:

```
import datetime

current_datetime = datetime.datetime.now()

print(current_datetime)
```

The `datetime.datetime.now()` function returns the current date and time as a `datetime.datetime` object.

When you print `current_datetime`, it will be displayed in the format `YYYY-MM-DD HH:MM:SS.ssssss`, where `YYYY` is the year, `MM` is the month, `DD` is the day, `HH` is the hour (in 24-hour format), `MM` is the minute, `SS` is the second, and `ssssss` represents microseconds.

To get the current date and time with a specific format in Python, you can use the `datetime` module along with the `strftime()` method. The `strftime()` method allows you to format the `datetime` object into a custom string representation based on specific format codes.

Here's an example of how to get the current date and time with a specific format:

```
import datetime

current_datetime = datetime.datetime.now()

# Format the datetime object into a custom string representation
formatted_datetime = current_datetime.strftime('%Y-%m-%d %H:%M:%S')

print(formatted_datetime)
```

In this example, `%Y`, `%m`, `%d`, `%H`, `%M`, and `%S` are format codes representing year, month, day, hour, minute, and second, respectively. When you use `strftime()` with these format codes, it will replace them with the corresponding values from the `current_datetime`.

You can customize the format string as per your requirement. Here are some commonly used format codes:

`%Y`: Year with century as a decimal number.

`%m`: Month as a zero-padded decimal number.

`%d`: Day of the month as a zero-padded decimal number.

`%H`: Hour (24-hour clock) as a zero-padded decimal number.

`%M`: Minute as a zero-padded decimal number.



%S: Second as a zero-padded decimal number.

You can include other characters or separators in the format string, which will be retained as they are in the output. For example, if you want to include a forward slash between the date components, you can use `'/%m/%d/%Y %H:%M:%S'` as the format string.

Remember to refer to the official Python documentation for more information on the available format codes and their meanings: <https://docs.python.org/3/library/datetime.html#strftime-strptime-behavior>

Practice

```
# file handling
# Read file
# Write data to file
# append data to file
# delete file
# move file
# directory
# create directory

import os
import shutil
import datetime

# Example 1
# home_dir = os.path.expanduser('~')
file_path = os.path.join(os.path.expanduser('~'),
                          'Desktop/file/')
print("file_path", file_path)
file_name = file_path + "test.txt"

def read_file():
    print(file_name)
    file = open(file_name, "r")
    for data in file:
        print(data)
```

```
file.close()

def read_file_using_func():
    # Example 2
    print("read data from file using read
function")
    file = open(file_name, "r")
    print(file.read())
    file.close()

# Example 3
def read_file_as():
    print("read file by using with and as file")
    with open(file_name) as file:
        data = file.read()
        print(data)

def read_file_lines():
    print("read file by using with and as file")
    with open(file_name) as file:
        data = file.readlines()
        print(type(data))
        for line in data:
            print(line)

file_name = "test.txt"

def write_data():
    file = open(file_name, "w")
    file.write("This is the command to write data
to text file")
    file.write("This is the command to write data
to text ")
```

```
    file.write("This is the command to write data
")
    file.close()

def append_data():
    file = open(file_name, "a")
    file.write("\nappend data to the existing
file")
    file.close()

file_name = "data.txt"

def write_data_with():
    with open(file_name, "w") as file:
        file.write("This is the command to write
data to text file\n")
        file.write("This is the command to write
data to text\n")
        file.write("This is the command to write
data\n")

def delete_file():
    # Delete file
    # os.remove(file_name)
    file_to_delete = "abc.txt"
    if os.path.exists(file_to_delete):
        os.unlink(file_to_delete)
    else:
        print("File not found")

test_file = "test.txt"
current_date_time = datetime.datetime.now()
current_date = datetime.date.today()
```

```
print("current_date", current_date)
formatted_date_time =
current_date_time.strftime('%Y-%m-%d %H-%M-%S')
print("formatted_date_time", formatted_date_time)
print("type current_date", type(current_date))

print(current_date_time)
print(type(current_date_time))
new_file_name = "test-" + str(formatted_date_time) +
".txt"
print("file name ", test_file)

print("new_file_name ", new_file_name)

def move_file():
    if os.path.exists(new_file_name):
        os.rename(test_file, new_file_name)
        print("file exist in the directory")
        shutil.move(new_file_name, file_path)
        print("file successfully move to the
destination directory", file_path)
    else:
        print("file not exist in the directory")

# move_file()

def read_all_files():
    file_names = os.listdir(file_path)
    print(file_names)
    for file_name in file_names:
        # print(file_name)
        desktop_file_path = os.path.join(file_path,
file_name)
        print(desktop_file_path)
```



```
        with open(desktop_file_path, "r") as file:
            print(file)

# read_all_files()
import glob

def read_all_files_glob():
    file_paths = glob.glob(os.path.join(file_path,
"*.*txt"))
    print(file_paths)
    for file_name in file_paths:
        if os.path.isfile(file_name):
            with open(file_name, "r") as file:
                content = file.read()
                print(file_name)
                print(content)

read_all_files_glob()
```