Module 5
Files handling and Regular Expressions

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.

In this article we will explore ***Python File Handling, Advantages, Disadvantages and How open, write and append functions works in python file.***

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.

For this article, we will consider the following "**geeks.txt**" file as an example.

```
Hello world
GeeksforGeeks
123 456
```

File Modes

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.

Reading from a text file

There is more than one way to How to read from a file in Python. Let us see how we can read the content of a file 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.

Method1:

```
# 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)
```

Method2

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

print(data)
```

Method3

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

Output:

```
Hello world
GeeksforGeeks
123 456
```

writing a text file

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.

Method1

```
# 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()
```

```
Method2

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

method 3

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

OUTPUT:

```
Hello World!!!
```

Reading binary files

Generally, binary means two. In computer science, 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. In this article, let's understand the reading of binary files.

To read a binary file,

Step 1: Open the binary file in binary mode

To read a binary file in Python, first, we need to open it in binary mode (‘”rb”‘). We can use the ‘open()’ function to achieve this.

Step 3: Read the binary data

Step 4: Process the binary data

Step 5: Close the file

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()

# Knowing the Type of our data
print(type(data))

# Printing our byte sequenced data
print(data)

# Closing the opened file
f.close()
```

Write Bytes to File

Files are used in order to store data permanently. File handling is performing various operations (read, write, delete, update, etc.) on these files. In Python, file handling process takes place in the following steps:

1. Open file
2. Perform operation
3. Close file

Example:

```
some_bytes = b'\x21'

# Open file in binary write mode
binary_file = open("my_file.txt", "wb")

# Write bytes to file
binary_file.write(some_bytes)

# Close file
binary_file.close()
```

Seek() Function:

The seek function is a built-in function in Python that is used to set the current position of the file pointer within a file. The file pointer is a marker that indicates the current position in the file, and it is used to read or write data from that point. This function is particularly useful when reading or writing large files, as it allows us to move the file pointer to a specific location within the file and read or write data from that point.

Syntax

```
file.seek(offset, whence)
```

The first argument, offset, is the number of bytes we want to move the file pointer. The second argument, whence, specifies the reference position from where we want to move the file pointer. The possible values of whence are

- 0 (default): refers to the beginning of the file
- 1: refers to the current position of the file pointer
- 2: refers to the end of the file

Example

data.txt File:

```
PrepBytes is an Ed-Tech Company.
# Open a file in read mode
file = open("data.txt", "r")

# Set the position of the file pointer to byte 10
file.seek(10)

# Read the next 5 bytes from the file
data = file.read(5)

# Print the data that was read
print(data)

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

Output:

```
is an
```

Python Regular Expression (Regex)

Essentially, a Python regular expression is a sequence of characters, that defines a search pattern. We can then use this pattern in a string-searching algorithm to "find" or "find and replace" on strings. You would've seen this feature in Microsoft Word as well.

In this Python Regex tutorial, we will learn the basics of regular expressions in Python. For this, we will use the 're' module. Let's import it before we begin.

```
1. >>> import re
```

Python Regex – Metacharacters

Each character in a Python Regex is either a metacharacter or a regular character. A metacharacter has a special meaning, while a regular character

Metacharacter	Description
^	Matches the start of the string
.	Matches a single character, except a newline But when used inside square brackets, a dot is matched
[ ]	A bracket expression matches a single character from the ones inside it [abc] matches 'a', 'b', and 'c' [a-z] matches characters from 'a' to 'z' [a-cx-z] matches 'a', 'b', 'c', 'x', 'y', and 'z'
[^]	Matches a single character from those except the ones mentioned in the brackets[^abc] matches all characters except 'a', 'b' and 'c'
( )	Parentheses define a marked subexpression, also called a block, or a capturing group
\t, \n, \r, \f	Tab, newline, return, form feed
*	Matches the preceding character zero or more times ab*c matches 'ac', 'abc', 'abbc', and so on [ab]* matches '', 'a', 'b', 'ab', 'ba', 'aba', and so on (ab)* matches '', 'ab', 'abab', 'ababab', and so on

{m,n}	Matches the preceding character minimum m times, and maximum n times a{2,4} matches ‘aa’, ‘aaa’, and ‘aaaa’
{m}	Matches the preceding character exactly m times
?	Matches the preceding character zero or one times ab?c matches ‘ac’ or ‘abc’

+	Matches the preceding character one or one times ab+c matches ‘abc’, ‘abbc’, ‘abbbc’, and so on, but not ‘ac’
|	The choice operator matches either the expression before it, or the one after abc|def matches ‘abc’ or ‘def’
\w	Matches a word character (a-zA-Z0-9) \W matches single non-word characters
\b	Matches the boundary between word and non-word characters
\s	Matches a single whitespace character \S matches a single non-whitespace character

	Examples- \. \\ \" When unsure if a character has a special meaning, put a \ before it: \@
$	A dollar matches the end of the string

A raw string literal does not handle backslashes in any special way. For this, prepend an ‘r’ before the pattern. Without this, you may have to use ‘\\\\’ for a single backslash character. But with this, you only need r’\’.

Regular characters match themselves

Rules for a Match

So, how does this work? The following rules must be met:

1. The search scans the string start to end.
2. The whole pattern must match, but not necessarily the whole string.
3. The search stops at the first match.

If a match is found, the group() method returns the matching phrase. If not, it returns None.

```
1. >>> print(re.search('na','no'))
```

None

Python Regular Expression Functions

We have a few functions to help us use Python regex.

a. match()

match() takes two arguments- a pattern and a string. If they match, it returns the string. Else, it returns None. Let's take a few Python regular expression match examples.

```
>>> print(re.match('center','centre'))
```

None

```
>>> print(re.match('...\w\we','centre'))
```

<_sre.SRE_Match object; span=(0, 6), match='centre'>

b. search()

search(), like match(), takes two arguments- the pattern and the string to

```
>>> match=re.search('aa?yushi','ayushi')
>>> match.group()
```

'ayushi'

```
>>> match=re.search('aa?yushi?','ayush ayushi')
>>> match.group()
```

'ayush'

```
>>> match=re.search('\w*end','Hey! What are your plans for the weekend?')
>>> match.group()
```

'weekend'

```
>>> match=re.search('^\w*end','Hey! What are your plans for the weekend?')
>>> match.group()
```

Traceback (most recent call last):

File "<pyshell#337>", line 1, in <module>

match.group()

AttributeError: 'NoneType' object has no attribute 'group'

Here, an AttributeError raised because it found no match. This is because we specified that this pattern should be at the beginning of the string. Let's try searching for space.

```
>>> match=re.search('i\sS','Ayushi Sharma')
>>> match.group()
```

'i S'

```
>>> match=re.search('\w+c{2}\w*','Occam\'s Razor')
>>> match.group()
```

'Occam'

It really will take some practice to get it into habit what the metacharacters mean. But since we don't have so many, this will hardly take an hour.

Python Regex Options

The functions we discussed may take an optional argument as well. These options are:

a. Python Regular Expression IGNORECASE

This Python Regex ignore case ignores the case while matching.

```
>>> match=re.findall(r'hi','Hi, did you ship it, Hillary?',re.IGNORECASE)
>>> for i in match:
print(i)
```

Hi

hi

b. Python MULTILINE

Working with a string of multiple lines, this allows ^ and $ to match the start and end of each line, not just the whole string.

```
>>> match=re.findall(r'^Hi','Hi, did you ship it, Hillary?\nNo, I didn\'t, but Hi',re.MULTILINE)
>>> for i in match:
print(i)
```

Hi

c. Python DOTALL

.* does not scan everything in a multiline string; it only matches the first line. This is because . does not match a newline. To allow this, we use DOTALL.

```
>>> match=re.findall(r'.*','Hi, did you ship it, Hillary?\nNo, I didn\'t, but Hi',re.DOTALL)
>>> for i in match:
print(i)
```

Hi, did you ship it, Hillary?

No, I didn't, but Hi

Python RegEx Functions

The Python 're' module offers a set of functions used to work with regular expressions and search a string for a matching pattern. Here are a few main functions we use:

Function	**Description**
re.findall(pattern,string)	The findall() function matches all the patterns in the string and returns the list with the matches.
re.search(pattern,string)	This function takes a regular expression and a string. It searches for and matches the pattern present at any position in the string. If the pattern is found, search() returns the matching object; else, it doesn't return any. It matches the first occurrence of the pattern.
re.compile()	This function compiles regular expressions into pattern objects for repeated use. It is useful for improving performance when the pattern is repeated.
re.escape(string)	This function escapes all special characters in a string and treats them as literals.

re.split(pattern,string)	This function splits the string and returns a list showing where the string has been split at each match.
re.sub(pattern,rep_substring,string)	It replaces one or more matching patterns in the string with the substring.

Here is a look at the definitions and examples of RegEx functions.

re.findall()

This function returns non-overlapping matches of patterns in a string and returns them in the order they are found. Strings are scanned left to right.

Example

```
import re

# Define the pattern to match email addresses
pattern = r'\b\w+@\w+\.\w+\b'

# Test string containing multiple email addresses
text = """
Contact us at support@example.com or sales@example.org.
For queries, email us at info@example.co or feedback@example.net.
"""

# Find all email addresses in the text
emails = re.findall(pattern, text)

# Print the result
print("Email addresses found:")
for email in emails:
    print(email)
```

Output:

```
Email addresses found:
support@example.com
sales@example.org
info@example.co
feedback@example.net
```

re.search()

This function matches the first occurrence of the pattern in a string. Therefore, it stops after finding the first match, which makes it best for testing regular expressions rather than extracting data. It returns re.MatchObject with all the information about the matched part or returns None if the pattern is not found.

Example

```
import re

# Define the pattern to match email addresses
pattern = r'\b\w+@\w+\.\w+\b'

# Test string containing multiple email addresses
text = """
Contact us at support@example.com or sales@example.org.
For queries, email us at info@example.co or feedback@example.net.
"""

# Search for the first email address in the text
match = re.search(pattern, text)

# Check if a match was found and print the result
if match:
    print("First email address found:", match.group())
else:
    print("No email address found.")
```

Output

```
First email address found: support@example.com
```

re.compile()

It combines RegEx patterns into RegEx objects with methods for different operations, such as performing string substitutions and searching for pattern matches.

Example

```
import re

# Compile the regular expression pattern into a Pattern object
pattern = re.compile(r'(\d{3})-(\d{2})-(\d{4})')

# Test string containing SSNs
text = """
John's SSN is 123-45-6789 and Jane's SSN is 987-65-4321.
Contact us at 555-55-5555 for more information.
"""

# Search for the first SSN in the text
search_match = pattern.search(text)
if search_match:
    print("First SSN found:", search_match.group())

# Find all SSNs in the text
all_matches = pattern.findall(text)
print("All SSNs found:", all_matches)
```

```
# Replace SSNs with a placeholder
replaced_text = pattern.sub('XXX-XX-XXXX', text)
print("Text with SSNs replaced:", replaced_text)
```

Output:

```
First SSN found: 123-45-6789
All SSNs found: [('123', '45', '6789'), ('987', '65', '4321'), ('555', '55',
'5555')]
Text with SSNs replaced:
John's SSN is XXX-XX-XXXX and Jane's SSN is XXX-XX-XXXX.
Contact us at XXX-XX-XXXX for more information.
```

re.escape()

This function returns all non-alphanumeric characters in a string. We use it to match an arbitrary literal string with RegEx metacharacters.

Syntax

```
re.escape(string)
```

Example

```
import re
# Define the exact string to search for
search_string = 'example.com/123?query=abc'
# Escape special characters in the search string
escaped_string = re.escape(search_string)
# Compile the escaped pattern
pattern = re.compile(escaped_string)
# Test string containing the search string
text = """
For more information, visit example.com/123?query=abc.
You can also go to example.com/456?query=xyz for different results.
"""
# Search for the exact string in the text
search_match = pattern.search(text)
if search_match:
    print("Exact match found:", search_match.group())
else:
    print("No exact match found.")
```

Output:

```
Exact match found: example.com/123?query=abc
```

re.split()

The Python RegEx split function splits the string by the pattern occurrences. Once the pattern is found, it returns the remaining characters from the string as a list after splitting.

Syntax

```
re.split(pattern, string, maxsplit=0, flags=0)
```

Here, pattern is the regular expression, string is where the pattern is searched and split occurs, maxsplit is zero if not provided, and if there is a non-zero value, then that many splits occur. Flags make the code short but are not necessary parameters.

Example

```
import re

# Define the pattern to match commas, semicolons, or spaces
pattern = r'[;,\s]+'

# Define the input string
text = "apple, banana; orange grape; pear, kiwi"

# Split the string using the pattern
items = re.split(pattern, text)

# Print the result
print("Items:", items)
```

Output:

```
Items: ['apple', 'banana', 'orange', 'grape', 'pear', 'kiwi']
```

re.sub()

The ‘sub’ denotes SubString. It searches a certain RegEx pattern in the given string, and once found, it replaces the pattern with the given substring in that specific string using repl. It also counts the number of checks and maintains the number of times it happens.

Syntax

```
 re.sub(pattern, repl, string, count=0, flags=0)
```

Example

```
import re

# Define the pattern to match dates in MM-DD-YYYY format
pattern = r'\b\d{2}-\d{2}-\d{4}\b'

# Define the replacement string
replacement = 'DATE'

# Define the input string
text = "John's birthday is 12-31-2023 and Jane's birthday is 01-01-2024."

# Replace all occurrences of the pattern with the replacement string
new_text = re.sub(pattern, replacement, text)

# Print the result
print("Modified text:", new_text)
```

Output:

```
Modified text: John's birthday is DATE and Jane's birthday is DATE.
```

re.subn()

The subn() function and sub() functions are the same. However, they differ in the way they provide output. Unlike sub() that returns only a string, the subn() function returns a tuple with total replacement counts and a new string.

Syntax

```
 re.subn(pattern, repl, string, count=0, flags=0)
```

Example

```
import re

# Define the pattern to match dates in MM-DD-YYYY format
pattern = r'\b\d{2}-\d{2}-\d{4}\b'

# Define the replacement string
replacement = 'DATE'

# Define the input string
text = "John's birthday is 12-31-2023 and Jane's birthday is 01-01-2024."

# Replace all occurrences of the pattern and count replacements
new_text, num_replacements = re.subn(pattern, replacement, text)

# Print the result
print("Modified text:", new_text)
print("Number of replacements:", num_replacements)
```

Output:

```
Modified text: John's birthday is DATE and Jane's birthday is DATE.
Number of replacements: 2
```