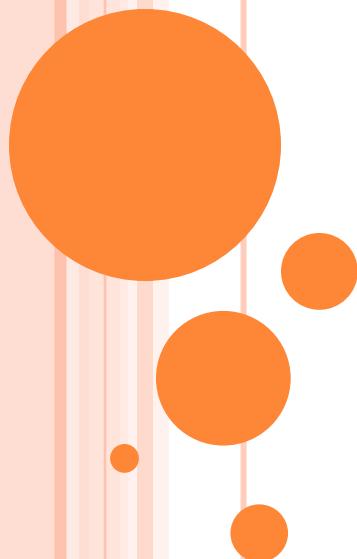


# UNIT-II

## FILES



# FILE

- **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 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.



# FUNCTIONS FOR FILE HANDLING

Method name	Description
open()	Open a file
read()	Read a file returns the whole text, but it can also specify how many characters you want to return
readline()	return one line
readlines()	Returns all the lines at a time
write()	Write contents in the file
close()	Close the file
seek()	to change the current file cursor position
tell()	to determine the current position of the file cursor.



# BASIC FILE OPERATIONS

- Opening a File

To open a file, use the `open()` function, which returns a file object. The function takes two main arguments: the file name and the mode in which to open the file.

Example:-

```
file = open('example.txt', 'r') # 'r' is for reading  
(default)
```



## With statement

- with statement in Python is used in exception handling to make the code cleaner and much more readable.
- It simplifies the management of common resources like file streams.
- Unlike the above implementations, there is no need to call `file.close()` when using with statement.
- The with statement itself ensures proper acquisition and release of resources.

**Syntax: with open filename as file:**

Example

```
with open("myfile.txt", "w") as file1:
```



# COMMON MODES:

1. **'r'**: Read (default mode). Opens the file for reading, throws an error if the file does not exist.
2. **'w'**: Write. Opens the file for writing, creates the file if it does not exist, and truncates the file if it exists.
3. **'a'**: Append. Opens the file for appending, creates the file if it does not exist.
4. **'b'**: Binary mode (can be combined with other modes, e.g., 'rb' or 'wb').
5. **r+:** To read and write data into the file. This mode does not override the existing data, but you can modify the data starting from the beginning of the file.
6. **w+:** To write and read data. It overwrites the previous file if one exists, it will truncate the file to zero length or create a file if it does not exist.
7. **a+:** To append and read data from the file. It won't override existing data.

8. **'rb'**: Read. Opens the file for reading in binary mode, throws an error if the file does not exist.
9. **'wb'**: Write. Opens the file for writing in binary mode, creates the file if it does not exist, and truncates the file if it exists.
10. **'ab'**: Append. Opens the file for appending in binary mode, creates the file if it does not exist.
11. **rb+:** To read and write data into the file in binary mode. This mode does not override the existing data, but you can modify the data starting from the beginning of the file.
12. **wb+:** To write and read data in binary mode. It overwrites the previous file if one exists, it will truncate the file to zero length or create a file if it does not exist.
13. **ab+:** To append and read data from the file in binary mode. It won't override existing data.
14. **x:** Create a new empty file, return error if the file already exists.

- o **Read a file:-**

There are three ways to read data from a text file.

1. **read()** : Returns the read bytes in form of a string. Reads n bytes, if no n specified, reads the entire file.

Syntax:- File\_object.read([n])

2. **readline()** : Reads a line of the file and returns in form of a string. For specified n, reads at most n bytes. However, does not reads more than one line, even if n exceeds the length of the line.

Syntax:- File\_object.readline([n])

3. **readlines()** : Reads all the lines and return them as each line a string element in a list.

Syntax:- File\_object.readlines()



```
# Write some data to a file
with open('example.txt', 'w') as file:
    file.write('Hello, this is a line of text.\n')
    file.write('This is another line of text.\n')
```

```
# Read the entire file
with open('example.txt', 'r') as file:
    content = file.read()
    print("Read entire file:")
    print(content)
```

```
# Read the first 10 bytes
with open('example.txt', 'r') as file:
    content = file.read(10)
    print("\nRead first 10 bytes:")
    print(content)
```



```
# Read a single line  
with open('example.txt', 'r') as file:  
    line = file.readline()  
    print("\nRead a single line:")  
    print(line)
```

```
# Read all lines as a list  
with open('example.txt', 'r') as file:  
    lines = file.readlines()  
    print("\nRead all lines as a list:")  
    print(lines)
```



## **# Writing to a file**

```
file=open('example.txt', 'w')
file.write('Hello, this is a line of text.\n')
file.write('This is another line of text.\n')
print("Data written to the file.")
file.close()
```

## **# Open a file in write mode (this will create the file if it doesn't exist)**

```
with open('example.txt', 'w') as file:
    file.write('Hello, this is a line of text.\n')
    file.write('This is another line of text.\n')
print("Data written to the file.")
```

## **# Reading from a file**

```
with open('example.txt', 'r') as file:
    content = file.read()
print("File content:")
print(content)
```



## **# Appending to a file**

```
with open('example.txt', 'a') as file:  
    file.write('Appending this line to the file.\n')  
print("Data appended to the file.")
```

## **# Reading the file line by line**

```
with open('example.txt', 'r') as file:  
    for line in file:  
        print(line, end='')  
    print("File read line by line.")
```



## **#seek() function**

with open('example.txt', 'r') as file:

```
    file.seek(5) # Move the cursor to the 6th byte
```

```
    content = file.read()
```

```
    print(content)
```

## **#tell() function**

with open('example.txt', 'r') as file:

```
    print(file.tell()) # Prints the current cursor position
```

```
    file.read(10)
```

```
    print(file.tell()) # Prints the new cursor position after  
    reading 10 bytes
```



```
# Delete the file
import os
# Create a sample file to demonstrate deletion
with open('sample.txt', 'w') as file:
    file.write('This is a sample file.')
```

```
# Delete the file
os.remove('sample.txt')
print("File 'sample.txt' deleted.")
import os
```

OR

```
# Delete the file
os.unlink('sample.txt')
print("File 'sample.txt' deleted.")
```



# DIRECTORIES

- Directories are a way of storing, organizing, and separating the files on a computer.
- The directory that does not have a parent is called a **root directory**.
- The way to reach the file is called the **path**.
- The path contains a combination of directory names, folder names separated by slashes and colon and this gives the route to a file in the system.
- Python contains several modules that has a number of built-in functions to manipulate and process data.
- Python has also provided modules that help us to interact with the operating system and the files.
- These kinds of modules can be used for directory management also.



# OS AND OS.PATH MODULE

- The os module is used to handle files and directories in various ways.
- It provides provisions to create/rename/delete directories.
- This allows even to know the current working directory and change it to another.
- It also allows one to copy files from one directory to another.



## Creating new directory:

- **os.mkdir(name)** method to create a new directory.
- The desired name for the new directory is passed as the parameter.
- By default it creates the new directory in the current working directory.
- If the new directory has to be created somewhere else then that path has to be specified and the path should contain forward slashes instead of backward ones.

Example:

```
import os  
  
# creates in current working directory  
  
os.mkdir('new_dir')
```

```
# creates in D:\  
  
os.mkdir('D:/new_dir')
```



## Getting Current Working Directory (CWD):

- **os.getcwd()** can be used.
- It returns a string that represents the path of the current working directory.
- **os.getcwdb()** can also be used but it returns a byte string that represents the current working directory.
- Both methods do not require any parameters to be passed.

Example:

```
import os  
  
print("String format :", os.getcwd())  
  
print("Byte string format :", os.getcwdb())
```



## **Renaming a directory:**

- **os.rename()** method is used to rename the directory.
- The parameters passed are old\_name followed by new\_name.
- If a directory already exists with the new\_name passed, OSError will be raised in case of both Unix and Windows.
- If a file already exists with the new\_name, in Unix no error arises, the directory will be renamed. But in Windows the renaming won't happen and error will be raised.
- **os.renames('old\_name','dest\_dir:/new\_name')** method works similar to **os.rename()** but it moves the renamed file to the specified destination directory(dest\_dir).

Example:

```
import os  
  
os.rename('file1.txt','file1_renamed.txt')
```

## **Changing Current Working Directory (CWD):**

- Every process in the computer system will have a directory associated with it, which is known as Current Working Directory(CWD).
- **os.chdir()** method is used to change it.
- The parameter passed is the path/name of the desired directory to which one wish to shift.

Example:

```
import os  
print("Current directory :", os.getcwd())  
# Changing directory  
os.chdir('/home/dyp/Desktop/')
```



```
print("Current directory :", os.getcwd())
```

- **Listing the files in a directory**
  - A directory may contain sub-directories and a number of files in it. To list them, **os.listdir()** method is used.
  - It either takes no parameter or one parameter.
  - If no parameter is passed, then the files and sub-directories of the CWD is listed.
  - If files of any other directory other than the CWD is required to be listed, then that directory's name/path is passed as parameter.

Example:

```
import os  
print("The files in CWD are :",os.listdir(os.getcwd()))
```



## Removing a directory

- **os.rmdir()** method is used to remove/delete a directory.
- The parameter passed is the path to that directory.
- It deletes the directory if and only if it is empty, otherwise raises an OSError.

Example:

```
import os  
dir_li=os.listdir('k:/files')  
if len(dir_li)==0:  
    print("Error!! Directory not empty!!")  
else:  
    os.rmdir('k:/files')
```



- **To check whether it is a directory:**
  - Given a directory name or Path, **os.path.isdir(path)** is used to validate whether the path is a valid directory or not.
  - It returns boolean values only. Returns **true** if the given path is a valid directory otherwise **false**.

Example:

```
import os

# current working directory of

# GeeksforGeeks

cwd=''

print(os.path.isdir(cwd))

# Some other directory

other='K:/'

print(os.path.isdir(other))
```



# MODULES

Modules in Python are files containing Python code (functions, classes, variables, etc.) that can be imported and used in other Python programs. They help in organizing and reusing code efficiently.

There are different types of modules:

1. **Standard Library Modules:** Built-in modules that come with Python.
2. **Third-Party Modules:** Modules you can install using package managers like pip.
3. **User-Defined Modules:** Custom modules created by users.



- o Creating and Using a User-Defined Module

1. **Creating a Module:** To create a Python module, write the desired code and save that in a file with .py extension

```
# A simple module, calc.py
```

```
def add(x, y):  
    return (x+y)  
  
def subtract(x, y):  
    return (x-y)
```



- Importing Modules

import modules using the import statement.

2. **# importing module calc.py**

3. import calc

4. print(calc.add(10, 2))

5. print(calc.subtract(10, 2))



## PACKAGE

- A package is a way of organizing related modules into a single directory hierarchy.
- Packages allow you to structure your code better and make it easier to manage large codebases.
- A package is simply a directory that contains a special file called `__init__.py` (which can be empty) and one or more module files.



1. Create the Package Directory Structure
2. Write the Modules
3. Use the created Package

Example:

```
mypackage/  
    __init__.py
```

```
        module1.py  
        module2.py
```

```
main.py
```



```
# mypackage/module1.py
def func1():
    return "This is function 1 from module 1."
```

```
# mypackage/module2.py
def func2():
    return "This is function 2 from module 2."
```



```
# main.py
import mypackage

print(mypackage.func1()) # Output: This is function 1 from module 1.
print(mypackage.func2()) # Output: This is function 2 from module 2.

from mypackage import module1, module2

print(module1.func1()) # Output: This is function 1 from module 1.
print(module2.func2()) # Output: This is function 2 from module 2.

from mypackage.module1 import func1
from mypackage.module2 import func2

print(func1()) # Output: This is function 1 from module 1.
print(func2()) # Output: This is function 2 from module 2.
```



# TEXT PROCESSING

- Text processing in Python involves manipulating and analyzing text data.
- Python offers a wide array of built-in functions, libraries, and tools to perform text processing tasks such as string manipulation, regular expressions, and natural language processing.
- Text Processing is an essential task in NLP as it helps to clean and transform raw data into a suitable format used for analysis or modeling.
- NLTK (Natural Language Toolkit) library is used for NLP.



# INSTALLING NLTK PACK

- `pip install nltk`
- “**Preferred Installer Program**” or PIP Installs Packages.
- It is a command-line utility that installs, reinstalls, or uninstalls PyPI packages with one simple command: `pip`
- `pip` is the **package installer** for Python.



# TOKENIZATION

- In Python tokenization basically refers to splitting up a larger body of text into smaller lines, words or even creating words for a non-English language.
- The various tokenization functions in-built into the nltk module itself.



- Tokenization: Tokenization is the process of splitting text into individual words or tokens.

```
from nltk.tokenize import word_tokenize, sent_tokenize  
# Word Tokenization  
text = "Hello, world! How are you?"  
words = word_tokenize(text)  
print(words) # Output: ['Hello', ',', 'world', '!', 'How',  
'are', 'you', '?']  
  
# Sentence Tokenization  
sentences = sent_tokenize(text)  
print(sentences) # Output: ['Hello, world!', 'How are  
you?']
```



# LINE TOKENIZATION

- In the below example divide a given text into different lines by using the function sent\_tokenize.

For Example:

```
import nltk  
  
sentence_data = "The First sentence is about Python. The Second:  
about Django. You can learn Python,Django and Data Ananlysis  
here. " nltk_tokens = nltk.sent_tokenize(sentence_data)  
  
print (nltk_tokens)
```



# WORD TOKENIZATION

- Tokenize the words using word\_tokenize function available as part of nltk.

For Example:

```
import nltk
```

```
word_data = "It originated from the idea that there are readers  
who prefer learning new skills from the comforts of their  
drawing rooms"
```

```
nltk_tokens = nltk.word_tokenize(word_data)
```

```
print (nltk_tokens)
```



## REMOVE DEFAULT STOPWORDS:

- Stopwords are words that do not contribute to the meaning of a sentence.
- Hence, they can safely be removed without causing any change in the meaning of the sentence.
- The NLTK library has a set of stopwords and can use these to remove stopwords from text and return a list of word tokens.



- Removing Stopwords: Stopwords are common words that are usually removed in text processing (e.g., "and", "the").

```
from nltk.corpus import stopwords  
# Download stopwords  
import nltk  
nltk.download('stopwords')  
  
stop_words = set(stopwords.words("english"))  
  
text = "This is a sample sentence, showing off the stop words filtration."  
words = word_tokenize(text)  
filtered_words = [word for word in words if word.lower() not in  
    stop_words]  
print(filtered_words) # Output: ['This', 'sample', 'sentence', ',', 'showing',  
    'stop', 'words', 'filtration', '.']
```

# STEMMING:

- Stemming is the process of getting the root form of a word.
- Stem or root is the part to which inflectional affixes (-ed, -ize, -de, -s, etc.) are added.
- The stem of a word is created by removing the prefix or suffix of a word.
- So, stemming a word may not result in actual words.



- If the text is not in tokens, then need to convert it into tokens.
- After converted strings of text into tokens, can convert the word tokens into their root form.
- There are mainly three algorithms for stemming.
- These are the **Porter Stemmer**, the **Snowball Stemmer** and the **Lancaster Stemmer**.
- Porter Stemmer is the most common among them.



- Stemming:

```
from nltk.stem import PorterStemmer  
ps = PorterStemmer()  
words = ["running", "runner", "ran", "runs"]  
stemmed_words = [ps.stem(word) for word in  
                 words]  
print(stemmed_words) # Output: ['run', 'runner',  
                           'ran', 'run']
```

# LEMMATIZATION:

- Like stemming, lemmatization also converts a word to its root form.
- The only difference is that lemmatization ensures that the root word belongs to the language.
- We will get valid words if we use lemmatization.
- In NLTK, we use the WordNetLemmatizer to get the lemmas of words.
- Need to provide a context for the lemmatization.
- So, we add the part-of-speech as a parameter.



```
from nltk.stem import WordNetLemmatizer  
# Download WordNet data  
nltk.download('wordnet')  
  
lemmatizer = WordNetLemmatizer()  
words = ["running", "runner", "ran", "runs"]  
lemmatized_words = [lemmatizer.lemmatize(word,  
    pos='v') for word in words]  
print(lemmatized_words) # Output: ['run', 'runner',  
    'run', 'run']
```

# REGULAR EXPRESSIONS

- A Regular Expression or RegEx is a **special sequence of characters** that uses a **search pattern** to find a string or set of strings.
- It can detect the presence or absence of a text by matching it with a particular pattern and also can split a pattern into one or more sub-patterns.
- Python has a **built-in module** named “**re**” that is used for regular expressions in Python.

*import re*



# METACHARACTERS

MetaCharacters	Description	Example
\	Used to drop the special meaning of character following it	"\d"
[ ]	Represent a character clas	"[a-m]"
^	Matches the beginning	"^hello"
\$	Matches the end	"planet\$"
.	Matches any character except newline	"he..o"
	Means OR (Matches with any of the characters separated by it.)	"falls stays"
?	Matches zero or one occurrence	"he.?o"
*	Any number of occurrences (including 0 occurrences)	"he.*o"
+	One or more occurrences	"he.+o"
{}	Indicate the number of occurrences of a preceding regex to match.	"he.{2}o"
()	Enclose a group of Regex	

# SPECIAL CHARACTERS

Special Sequence	Description	Example
\A	Matches if the string begins with the given character	\Afor for geeks for the world
\b	Matches if the word begins or ends with the given character. \b(string) will check for the beginning of the word and (string)\b will check for the ending of the word.	
\B	It is the opposite of the \b i.e. the string should not start or end with the given regex.	\Bge together forge
\d	Matches any decimal digit, this is equivalent to the set class [0-9]	
\D	Matches any non-digit character, this is equivalent to the set class [^0-9]	\D geeks geek1

# SPECIAL CHARACTERS

Special Sequence	Description		Example
\s	Matches any whitespace character.	\s	gee ks
			a bc a
\S	Matches any non-whitespace character	\S	a bd
			abcd
\w	Matches any alphanumeric character, this is equivalent to the class [a-zA-Z0-9_].	\w	123
			geeKs4
\W	Matches any non-alphanumeric character.	\W	>\$
			gee<>
\Z	Matches if the string ends with the given regex	ab\Z	abcdab
			abababab

# REGEX FUNCTIONS

Function	Description
re.findall()	finds and returns all matching occurrences in a list
re.compile()	Regular expressions are compiled into pattern objects
re.split()	Split string by the occurrences of a character or a pattern.
re.sub()	Replaces all occurrences of a character or pattern with a replacement string.
re.escape()	Escapes special character
re.search()	Searches for first occurrence of character or pattern



## 1. re.findall()

- Return all non-overlapping matches of pattern in string, as a list of strings.
- Finding all occurrences of a pattern

```
import re  
string = "Hello my Number is 123456789 and  
my friend's number is 987654321"  
regex = '\d+'  
match = re.findall(regex, string)  
print(match)
```

*Output*

`['123456789', '987654321']`



## 2. re.compile()

- Regular expressions are compiled into pattern objects, which have methods for various operations such as searching for pattern matches or performing string substitutions.
- **find and list all lowercase letters from ‘a’ to ‘e’**

*import re*

*p = re.compile('[a-e]')*

*print(p.findall("Aye, said Mr. Gibenson Stark"))*

*Output*

*['e', 'a', 'd', 'b', 'e', 'a']*



## 2. re.compile()

```
import re
```

```
p = re.compile('\d')
```

```
print(p.findall("I went to him at 11 A.M. on 4th July  
1886"))
```

```
p = re.compile('\d+')
```

```
print(p.findall("I went to him at 11 A.M. on 4th July  
1886"))
```

*Output*

```
['1', '1', '4', '1', '8', '8', '6']
```

```
['11', '4', '1886']
```



## 2. re.compile()

```
import re  
p = re.compile('ab*')  
print(p.findall("ababbaabbb"))
```

*Output*

```
['ab', 'abb', 'a', 'abbb']
```



### 3. re.split()

- Split string by the occurrences of a character or a pattern, upon finding that pattern, the remaining characters from the string are returned as part of the resulting list.
- Syntax :

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

- **pattern** denotes the regular expression,
- **string** is the given string in which pattern will be searched for and in which splitting occurs,
- **maxsplit** if not provided is considered to be zero ‘0’, and if any nonzero value is provided, then at most that many splits occur. If maxsplit = 1, then the string will split once only, resulting in a list of length 2.
- **The flags** are very useful and can help to shorten code, they are not necessary parameters, eg: flags = re.IGNORECASE, in this split, the case, i.e. the lowercase or the uppercase will be ignored.

### 3. re.split()

*from re import split*

```
print(split('\W+', 'Words, words , Words'))
print(split('\W+', "'Word's words Words'"))
print(split('\W+', 'On 12th Jan 2016, at 11:02 AM'))
print(split('\d+', 'On 12th Jan 2016, at 11:02 AM'))
```

*Output*

```
['Words', 'words', 'Words']
['Word', 's', 'words', 'Words']
['On', '12th', 'Jan', '2016', 'at', '11', '02', 'AM']
['On ', 'th Jan ', ', at ', ':', 'AM']
```



## 4. re.sub()

- The ‘sub’ in the function stands for SubString, a certain regular expression pattern is searched in the given string (3rd parameter), and upon finding the substring pattern is replaced by repl (2nd parameter), count checks and maintains the number of times this occurs.
- Syntax:

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

- Example:
- First statement **replaces all occurrences of ‘ub’ with ‘~\*’** (case-insensitive): 'S~\*ject has ~\*er booked already'.
- Second statement **replaces all occurrences of ‘ub’ with ‘~\*’** (case-sensitive): 'S~\*ject has Uber booked already'.
- Third statement replaces the **first occurrence of ‘ub’ with ‘~\*’** (case-insensitive): 'S~\*ject has Uber booked already'.
- Fourth replaces **‘AND’ with ‘ & ’** (case-insensitive): 'Baked Beans & Spam'.



## 4. re.sub()

```
import re  
print(re.sub('ub', '~*', 'Subject has Uber booked already',  
            flags=re.IGNORECASE))  
print(re.sub('ub', '~*', 'Subject has Uber booked already'))  
print(re.sub('ub', '~*', 'Subject has Uber booked already',  
            count=1, flags=re.IGNORECASE))  
print(re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam',  
            flags=re.IGNORECASE))
```

### Output

S~\*ject has ~\*er booked already  
S~\*ject has Uber booked already  
S~\*ject has Uber booked already  
*Baked Beans & Spam*



## 5. re.escape()

- Returns string with all non-alphanumerics backslashed, this is useful if you want to match an arbitrary literal string that may have regular expression metacharacters in it.
- `re.escape()` is used to escape special characters in a string, making it safe to be used as a pattern in regular expressions. It ensures that any characters with special meanings in regular expressions are treated as literal characters.

*`re.escape(string)`*



## 5. re.escape()

*import re*

*# Pattern*

*pattern = 'https://www.google.com/'*

*# Using escape function to escape metacharacters*

*result = re.escape( pattern)*

*# Printing the result*

*print('Result:', result)*

*Output*

*Result: https://www\.google\.com/*



## 6. re.search()

- This method either returns **None** (if the pattern doesn't match), or a `re.MatchObject` contains information about the **matching part of the string**. This method stops after the first match, so this is best suited for testing a regular expression more than extracting data.
- This code uses a regular expression to search for a pattern in the given string. If a match is found, it extracts and prints the matched portions of the string.



## 6. re.search()

```
import re
```

```
line = "Cats are smarter than dogs"
```

```
matchObj = re.search( r'than', line)
```

```
print (matchObj.start(), matchObj.end())
```

```
print ("matchObj.group() : ", matchObj.group())
```

```
print(res)
```

*Output*

*17 21*

*matchObj.group() : than*

