Files and Excepitons

1 What's a file?

- A file is a sequence of bytes stored on a secondary memory device, such as a disk drive.
- Files can be text documents, spreadsheets, HTML files, Python modules, executable applications, images, or audio files.
- Text files contain a sequence of characters that are encoded using some encoding, while binary files are just a sequence of bytes and have no encoding.

2 What's a file system?

- The file system is the component of a computer system that organizes files and provides ways to create, access, and modify files.
- The file system organizes files and folders into a tree structure, with the root directory at the top.
- Folders can contain other folders and regular files.
- Every file and folder in a file system has a name, but a name is not sufficient to locate a file efficiently. - - Instead, every file can be specified using a pathname.
  - The absolute pathname of a file consists of the sequence of folders, starting from the root directory, that must be traversed to get to the file.
  - The relative pathname of a file is the sequence of directories that must be traversed, starting from the current working directory, to get to the file.
- The double-period notation (`..`) is used to refer to the parent folder, which is the folder containing the current working directory.

3 Processing a file in python

Processing a file consists of these three steps: 1. Opening a file for reading or writing 2. Reading from the file and/or writing to the file 3. Closing the file ## 1. Opening a file Python introduces the `file` object in order to perform some file operations. Python has a built-in `open()` function to open files from the directory. Two arguments that are mainly needed by the `open()` function are:

- **file name** or **file path**: is a string that specifies the name of the file to be accessed.
- **access_mode**: parameter determines the mode in which the file will be opened, such as read, write, or append. We can also specify whether a file should be opened in the text mode or the binary mode (deals with bites in the case of non-text file). In the

Access mode	Description
'r'	Reading mode, file pointer is placed at the beginning of the file (default mode).
'rb'	Reading mode in binary format, file pointer is placed at the beginning of the file.
'r+'	Reading and writing mode, file pointer is placed at the beginning of the file.
'rb+'	Reading and writing mode in binary format, file pointer is placed at the beginning of the file.
'w'	Writing only mode, overwrites the existing file or creates a new file.
'wb'	Writing only mode in binary format, overwrites the existing file or creates a new file.
'w+'	Reading and writing mode, overwrites the existing file or creates a new file.
'wb+'	Reading and writing mode in binary format, overwrites the existing file or creates a new file.
'a'	Appending mode, file pointer is placed at the end of the file.
	If the file does not exist, a new file is created.
'ab'	Appending mode in binary format, file pointer is placed at the end of the file. If the file does not exist, a new file is created.
'a+'	Appending and reading mode, file pointer is placed at the end of the file. If the file does not exist, a new file is created.
'ab+'	Appending and reading mode in binary format, file pointer is placed at the end of the file. If the file does not exist, a new file is created.

3.0.1 The syntax for opening a file

The syntax for opening a file is:

```
file_object = open(file_name [, access_mode])
```

3.0.2 Examples:

```
f = open("test.txt") # opens in r mode(reading only)
f = open("test.txt",'w') # opens in w mode(writing only)
f = open("image.bmp",'rb+') # read and write in binary mode
```

3.1 2. Reading from the file

To read from a file in Python, you need to open the file in read mode. Once the file is opened, you can read its contents using various methods such as:

Method | Explanation :———————|:———————————————————- `fname.read(n)` | Read n characters from the file infile or until the end of the file is reached, and return characters read as a string `fname.read()` | Read characters from file infile until the end of the file and return characters read as a string `fname.readline()` | Read file infile until (and including) the new line character or until end of file, whichever is first, and return characters read as a string

`fname.readlines()` | Read file infile until the end of the file and return the characters read as a list lines

3.1.1 Examples

```
f=open("example.txt")
print(f.read(5))
#print(f.read())
print(f.readline())
print(f.readlines())
#f.close()
#f=open("example.txt")
#for line in f:
#        print(line)
```

3.2 3. Writing to the file

To write to a file in Python, you need to open the file in write mode using the built-in `open()` function with the `'w'` or `'wb'` mode depending on the file's content type. Here's an example:

```
file = open('filename.txt', 'w')
file.write('Hello, world!')
file.close()
```

This code opens a file named `'filename.txt'` in write mode using the `'w'` mode. Then, it writes the string `'Hello, world!'` to the file using the `write()` method. Finally, the file is closed using the `close()` method.

If you want to write multiple lines to the file, you can use the newline character `'\n'` to separate the lines, like this:

```
file = open('filename.txt', 'w')
file.write('Line 1\n')
file.write('Line 2\n')
file.close()
```

This code will write two lines to the file, each on a separate line.

Note that using the `'w'` mode will overwrite any existing content in the file. If you want to append to an existing file, you can use the `'a'` or `'ab'` mode instead of `'w'`.

We can also write many lines using `writelines(list_of_line)` method.

3.2.1 Line Endings

In Python, the new line character is represented by the escape sequence `\n` . However text file formats are platform dependent, and different operating systems use a different byte sequence to encode a new line: - MS Windows uses the `\r\n` 2-character sequence. - Linux/UNIX and Mac OS X use the `\n` character. - Mac OS up to version 9 uses the `\r` character. ## 4. Closing files In Python, it is important to close files after you have finished using them.

This releases the resources that were used by the file, and ensures that any data that was buffered in memory is written to the file. To close a file, you can use the `close()` method of the file object.

For example, if you opened a file using `open('example.txt', 'r')`, you can close it using `close()` method:

```
file_object = open('example.txt', 'r')
# Do some operations with the file
file_object.close()
```

It is recommended to always close files after you are done working with them, even though Python automatically closes the file when the program terminates or the file object is destroyed. # Attributes of a file object in Python A file object has several attributes that provide information about the file, including:

Attribute	Explanation
name	the name of the file
mode	the mode in which the file is opened
closed	a boolean indicating whether the file is closed or not
encoding	the encoding used to read or write the file (only applicable for text files)

[ ]:
```
f=open("example.txt")
print(f.name)
print(f.mode)
print(f.closed)
print(f.encoding)
```

4 File Positions

The `tell()` method of a file object returns the current position of the file pointer in bytes from the beginning of the file. It tells you where the next read or write operation will occur.

For example:

```
file = open("example.txt", "r")
print(file.tell()) # prints the current position of the file pointer
file.close()
```

The `seek(offset[, from])` method changes the current file position to the position specified by offset. The from argument is optional and specifies the reference position from where the bytes are to be moved. If from is not specified, it defaults to `0` (the beginning of the file).

For example:

```
file = open("example.txt", "r")
file.seek(10) # move the file pointer to the 10th byte from the beginning of the file
print(file.tell)
file.close()
```

```
#Binary files
file = open("example.txt", "rb")
file.read(10)
print(file.tell())
file.seek(10,1) # move the file pointer to the 10th byte from the current position of the file
print(file.tell())
file.read()
print(file.tell())
file.seek(-20,2) # move the file pointer 20 bytes backwards from the end of the file
print(file.tell())
file.close()
```

5 Renaming and Deleting Files

Python's os module provides functions to perform file-related operations such as renaming and deleting files. To use these functions, you need to import the os module first.

Example of renaming a file using `os.rename()`:

```
import os

# current file name
current_name = "old_file.txt"

# new file name
new_name = "new_file.txt"
# renaming the file
os.rename(current_name, new_name)
```

Example of deleting a file using `os.remove()`:

```
import os

# file name to be deleted
file_name = "file_to_delete.txt"

# deleting the file
os.remove(file_name)
```

6 Directories in Python

All files are contained within various directories, and Python has no problem handling these too. The `os` module has several methods that help you create, remove, and change directories.

6.1 The `mkdir()` Method

You can use the `mkdir()` method of the os module to create directories in the current directory. You need to supply an argument to this method which contains the name of the directory to be

created. For example, to create a new directory named `"new_folder"`, you can use the following code:

```
import os

os.mkdir("new_folder")
```

6.2 The `chdir()` Method

You can use the `chdir()` method to change the current directory. The `chdir()` method takes an argument, which is the name of the directory that you want to make the current directory. For example, to change the current directory to `"new_folder"`, you can use the following code:

```
import os

os.chdir("new_folder")
```

6.3 The `getcwd()` Method

The `getcwd()` method displays the current working directory. For example, to print the current working directory, you can use the following code:

```
import os

print(os.getcwd())
```

6.4 The `rmdir()` Method

The `rmdir()` method deletes the directory, which is passed as an argument in the method. Before removing a directory, all the contents in it should be removed. For example, to remove the `"new_folder"` directory, you can use the following code:

```
import os

os.rmdir("new_folder")
```

[1]:
```
file = open("example.txt", "rb")
file.read(10)
print(file.tell())
file.seek(10,1) # move the file pointer to the 10th byte from the current␣
 ↪position of the file
print(file.tell())
file.read()
print(file.tell())
file.seek(-20,2) # move the file pointer to the 10th byte from the current␣
 ↪position of the file
print(file.tell())
file.close()
```

```
10
20
```

```
679
659
```

6.4.1 open files using `with` statement

The basic syntax for opening a file using the `with` statement is as follows:

```
with open('filename.txt', 'r') as file:
    # do something with the file
```

In this example, we open the file `"filename.txt"` in read mode (`'r'`) using the `open()` function. The with statement creates a block of code where the file is open and available to use, and automatically closes the file when the block of code is exited.

This ensures that the file is properly closed even if an exception is raised within the block of code. Example:

```
with open('output.txt', 'a+') as file:
    file.write('Hello, world!')
```

7 Introduction to Exceptions and Error Handling

An exception is a signal that an error has occurred, and Python's built-in exception handling mechanism allows you to gracefully handle these errors and prevent your program from crashing.

7.1 Using `try-except` Blocks to Handle Exceptions

In Python, you can use the try and except statements to handle exceptions. The basic syntax is as follows:

```
try:
    # code that might raise an exception
except ExceptionType:
    # code to handle the exception
```

When the code inside the `try` block raises an exception (error), the interpreter jumps to the corresponding except block. The `ExceptionType` argument specifies the type of exception to catch. For example, to catch all exceptions, you can use the `Exception` base class:

```
try:
    # code that might raise an exception
except Exception:
    # code to handle the exception
```

Here's an example of how to use a `try-except` block to handle a ZeroDivisionError:

```
try:
    result = 1 / 0
except ZeroDivisionError:
    print("Are you serious! You can't divide by zero")
```

In this example, the `try` block attempts to divide 1 by 0, which raises a ZeroDivisionError. The corresponding except block catches the error and prints an error message.

7.2 Handling Multiple Types of Exceptions

You can also catch specific types of exceptions, such as TypeError or ValueError. To handle multiple types of exceptions, you can include multiple except blocks:

```
try:
    # code that might raise an exception
except TypeError:
    # code to handle a TypeError
except ValueError:
    # code to handle a ValueError
```

Here's an example of how to handle multiple types of exceptions:

```
try:
    result = int("not a number")
except (TypeError, ValueError):
    print("Error: Invalid input")
```

In this example, the try block attempts to convert the string `"not a number"` to an integer, which raises a `ValueError`. The corresponding except block catches both `TypeError` and `ValueError` exceptions and prints an error message. ## Multiple `except` and `else` block

In addition to handling different types of exceptions, you can also use multiple `except` and `else` to further customize the behavior of your error handling code.

When using multiple except blocks, you can specify different exception types and handle each one differently. Here's an example:

```
try:
    x = int(input("Enter a number: "))
    y = 10 / x
except ValueError:
    print("Invalid input")
except ZeroDivisionError:
    print("Cannot divide by zero")
else:
    print("Result is:", y)
```

7.3 Using `finally` Blocks

In addition to the try and except statements, Python provides the finally statement to run code that should execute regardless of whether an exception is raised:

```
try:
    # code that might raise an exception
except Exception:
    # code to handle the exception
finally:
    # code that always runs, even if there was an exception
```

Here's an example of how to use a finally block:

```
file = open("myfile.txt", "w")
try:
    # write data to the file
finally:
    file.close()
```

In this example, the try block writes data to a file, and the finally block ensures that the file is closed, even if an exception is raised.

7.4 Raising Exceptions

You can use the raise statement to raise an exception of a specified type, along with an optional error message:

```
raise ExceptionType("Error message")
```

Here's an example of how to raise a `ValueError`:

```
import sys
try:
    x=int(input("Give a positive integer > 19\n"))
    if (x<=19):
        raise ValueError("Invalid input Value")
except ValueError:
    print("Please focus a little!")
    sys.exit(1)
print("Continue !")
# rest of code
```

In this example, the my_function function checks whether the input is valid, and raises a `ValueError`.