Q1. Describe the differences between text and binary files in a single paragraph.

Text files are organized around lines, each of which ends with a newline character ('\n'). The source code files are themselves text files. A binary file is the one in which data is stored in the file in the same way as it is stored in the main memory for processing.

A text file is the one in which data is stored in the form of ASCII characters and is normally used for storing a stream of characters. Text files are organized around lines, each of which ends with a newline character (‘\n’). The source code files are themselves text files.

A binary file is the one in which data is stored in the file in the same way as it is stored in the main memory for processing. It is stored in binary format instead of ASCII characters. It is normally used for storing numeric information (int, float, double). Normally a binary file can be created only from within a program and its contents can be read only by a program.

Text File vs Binary File

The following are some of the differences between text files and binary files.

| S. No. | Text file | Binary File |
| --- | --- | --- |
| 1. | The text files can easily be transferred from one computer system to another. | Binary files cannot easily be transferred from one computer system to another due to variations in the internal variations in the internal representation which varies from computer to computer. |
| 2. | It stores data using ASCII format i.e. human-readable graphic characters. | It stores data in binary format i.e. with the help of 0 and 1. |
| 3. | These files are easily readable and modifiable because the content written in text files is human readable. Content written in binary files is not human-readable and looks like encrypted content. | These files are not easily readable and modifiable because the content written in binary files is not human-readable and it is encrypted content. |
| 4. | These files create portability problems. | These files are easily portable. |
| 5. | Text files save the data by converting each digit in data into ASCII format which will take up much of the space as compared to the required one.  For example, the number 546378 is an integer that should occupy 4 bytes in the disk but it will occupy 6 bytes, 1 byte for each digit in the number. | These save memory because the data of any type will get stored in memory as per its memory size.  For example, any integer number irrespective of individual digits in the number will be stored by consuming 4 bytes. |
| 6. | Any file is by default text file. | The ios:: binary mode has to be used with binary files while opening them. |
| 7. | Error in a textual file can be easily recognized and eliminated. | Error in a binary file corrupts the file and is not easily detected. |
| 8. | In a text file, a new line character is first converted to a carriage return-line feed combination and then written to the disk. Vice versa happens when a line is read from the text file. | In binary file, no such conversion from newline to carriage return-line feed combination is done. |
| 9. | In a text file, a special character with ASCII code 26 is inserted at the end of the file. This character signals the EOF to the program when encountered. | There is no such special character in the binary file to signal EOF. |

Q2. What are some scenarios where using text files will be the better option? When would you like to use binary files instead of text files?

On a computer, every file is a long string of ones and zeros. Specifically, a file is a finite-length sequence of bytes, where each [byte](https://en.wikipedia.org/wiki/Byte) is an integer between 0 and 255 inclusive (represented in binary as 00000000 to 11111111). Files can be broadly classified as either binary or text. These categories have different characteristics and need different tools to work with such files. Knowing the differences between binary and text files can save you time and mistakes when reading or writing data.

Here is the primary difference: [Binary files](https://en.wikipedia.org/wiki/Binary_file) have no inherent constraints (can be any sequence of bytes), and must be opened in an appropriate program that knows the specific file format (such as Media Player, Photoshop, Office, etc.). [Text files](https://en.wikipedia.org/wiki/Text_file) must represent reasonable text (explained later), and can be edited in any [text editor](https://en.wikipedia.org/wiki/Text_editor) program.

Remember that all files, whether binary or text, are composed of bytes. The difference between binary and text files is in how these bytes are interpreted. Every text file is indeed a binary file, but this interpretation gives us no useful operations to work with. The reverse is not true, and treating a binary file as a text file can lead to data corruption. As a method of last resort, a [hex editor](https://en.wikipedia.org/wiki/Hex_editor) can always be used to view and edit the raw bytes in any file.

File extensions

We can usually tell if a file is binary or text based on its file extension. This is because by convention the extension reflects the file format, and it is ultimately the file format that dictates whether the file data is binary or text.

Common extensions that are binary file formats:

Images: [jpg](https://en.wikipedia.org/wiki/JPEG), [png](https://en.wikipedia.org/wiki/Portable_Network_Graphics" \o "Wikipedia: Portable Network Graphics), [gif](https://en.wikipedia.org/wiki/GIF), [bmp](https://en.wikipedia.org/wiki/BMP_file_format), [tiff](https://en.wikipedia.org/wiki/Tagged_Image_File_Format), [psd](https://en.wikipedia.org/wiki/Adobe_Photoshop" \l "File_format" \o "Wikipedia: Adobe Photoshop - File format), ...

Videos: [mp4](https://en.wikipedia.org/wiki/MPEG-4_Part_14), [mkv](https://en.wikipedia.org/wiki/Matroska" \o "Wikipedia: Mastroska), [avi](https://en.wikipedia.org/wiki/Audio_Video_Interleave" \o "Wikipedia: Audio Video Interleave), [mov](https://en.wikipedia.org/wiki/QuickTime_File_Format), [mpg](https://en.wikipedia.org/wiki/MPEG-1), [vob](https://en.wikipedia.org/wiki/VOB" \o "Wikipedia: VOB), ...

Audio: [mp3](https://en.wikipedia.org/wiki/MP3), [aac](https://en.wikipedia.org/wiki/Advanced_Audio_Coding" \o "Wikipedia: Advanced Audio Coding), [wav](https://en.wikipedia.org/wiki/WAV), [flac](https://en.wikipedia.org/wiki/FLAC" \o "Wikipedia: FLAC), [ogg](https://en.wikipedia.org/wiki/Ogg" \o "Wikipedia: Ogg), [mka](https://en.wikipedia.org/wiki/Matroska" \o "Wikipedia: Mastroska), [wma](https://en.wikipedia.org/wiki/Windows_Media_Audio" \o "Wikipedia: Windows Media Audio), ...

Documents: [pdf](https://en.wikipedia.org/wiki/Portable_Document_Format), [doc](https://en.wikipedia.org/wiki/Doc_%28computing%29), [xls](https://en.wikipedia.org/wiki/Microsoft_Excel" \l "File_formats" \o "Wikipedia: Microsoft Excel - File formats), [ppt](https://en.wikipedia.org/wiki/Microsoft_PowerPoint#File_formats), [docx](https://en.wikipedia.org/wiki/Office_Open_XML), [odt](https://en.wikipedia.org/wiki/OpenDocument" \o "Wikipedia: OpenDocument), ...

Archive: [zip](https://en.wikipedia.org/wiki/Zip_%28file_format%29), [rar](https://en.wikipedia.org/wiki/RAR_%28file_format%29" \o "Wikipedia: RAR (file format)), [7z](https://en.wikipedia.org/wiki/7z), [tar](https://en.wikipedia.org/wiki/Tar_%28computing%29), [iso](https://en.wikipedia.org/wiki/ISO_image), ...

Database: [mdb](https://en.wikipedia.org/wiki/Microsoft_Access" \l "File_extensions" \o "Wikipedia: Microsoft Access - File extensions), [accde](https://en.wikipedia.org/wiki/Microsoft_Access" \l "File_extensions" \o "Wikipedia: Microsoft Access - File extensions), [frm](https://en.wikipedia.org/wiki/MySQL" \o "Wikipedia: MySQL), [sqlite](https://en.wikipedia.org/wiki/SQLite" \o "Wikipedia: SQLite), ...

Executable: [exe](https://en.wikipedia.org/wiki/.exe), [dll](https://en.wikipedia.org/wiki/Dynamic-link_library" \o "Wikipedia: Dynamic-link library), [so](https://en.wikipedia.org/wiki/Library_%28computing%29#Shared_libraries), [class](https://en.wikipedia.org/wiki/Java_class_file), ...

Common extensions that are text file formats:

Web standards: [html](https://en.wikipedia.org/wiki/HTML), [xml](https://en.wikipedia.org/wiki/XML), [css](https://en.wikipedia.org/wiki/Cascading_Style_Sheets" \o "Wikipedia: Cascading Style Sheets), [svg](https://en.wikipedia.org/wiki/Scalable_Vector_Graphics" \o "Wikipedia: Scalable Vector Graphics), [json](https://en.wikipedia.org/wiki/JSON" \o "Wikipedia: JSON), ...

Source code: [c](https://en.wikipedia.org/wiki/C_%28programming_language%29), [cpp](https://en.wikipedia.org/wiki/C%2B%2B" \o "Wikipedia: C++), [h](https://en.wikipedia.org/wiki/Include_directive#C.2FC.2B.2B), [cs](https://en.wikipedia.org/wiki/C_Sharp_%28programming_language%29), [js](https://en.wikipedia.org/wiki/JavaScript" \o "Wikipedia: JavaScript), [py](https://en.wikipedia.org/wiki/Python_%28programming_language%29" \o "Wikipedia: Python (programming language)), [java](https://en.wikipedia.org/wiki/Java_%28programming_language%29), [rb](https://en.wikipedia.org/wiki/Ruby_%28programming_language%29" \o "Wikipedia: Ruby (programming language)), [pl](https://en.wikipedia.org/wiki/Perl), [php](https://en.wikipedia.org/wiki/PHP" \o "Wikipedia: PHP), [sh](https://en.wikipedia.org/wiki/Shell_script" \o "Wikipedia: Shell script), ...

Documents: [txt](https://en.wikipedia.org/wiki/Text_file), [tex](https://en.wikipedia.org/wiki/TeX" \o "Wikipedia: TeX), [markdown](https://en.wikipedia.org/wiki/Markdown), [asciidoc](https://en.wikipedia.org/wiki/AsciiDoc" \o "Wikipedia: AsciiDoc), [rtf](https://en.wikipedia.org/wiki/Rich_Text_Format), [ps](https://en.wikipedia.org/wiki/PostScript" \o "Wikipedia: PostScript), ...

Configuration: [ini](https://en.wikipedia.org/wiki/INI_file" \o "Wikipedia: INI file), [cfg](https://en.wikipedia.org/wiki/Configuration_file" \o "Wikipedia: Configuration file), [rc](https://en.wikipedia.org/wiki/Run_commands" \o "Wikipedia: Run commands), [reg](https://en.wikipedia.org/wiki/Windows_Registry#.REG_files), ...

Tabular data: [csv](https://en.wikipedia.org/wiki/Comma-separated_values), [tsv](https://en.wikipedia.org/wiki/Tab-separated_values" \o "Wikipedia: Tab-separated values), ...

Binary file characteristics

Binary file in application (good)

Binary file in hex editor (okay)

Binary file in text editor (bad)

For most software that people use in their daily lives, the software consumes and produces binary files. Examples of such software include Microsoft Office, Adobe Photoshop, and various audio/video/media players. A typical computer user works with mostly binary files and very few text files.

A binary file always needs a matching software to read or write it. For example, an MP3 file can be produced by a sound recorder or audio editor, and it can be played in a music player or audio editor. But an MP3 file cannot be played in an image viewer or a database software.

Some binary formats are popular enough that a wide variety of programs can produce or consume it. Image formats like JPEG are the best example – not only can they be used in image viewers and editors, they can be viewed in web browsers, audio players (for album art), and document software (such as adding a picture into a Word doc). But other binary formats, especially for niche proprietary software, might have only one program in the world that can read and write it. For example, a high-end video editing software might let you save your project to a file, but this software is the only one that can understand its own file format; the binary file will never be useful anywhere else.

If you use a text editor to open a binary file, you will see copious amounts of garbage, seemingly random accented and Asian characters, and long lines overflowing with text – this exercise is safe but pointless. However, editing or saving a binary file in a text editor will corrupt the file, so never do this. The reason corruption happens is because applying a text mode interpretation will change certain byte sequences – such as discarding NUL bytes, converting newlines, discarding sequences that are invalid under a certain character encoding, etc. – which means that opening and saving a binary file will almost surely produce a file with different bytes.

Text file characteristics

Q3. What are some of the issues with using binary operations to read and write a Python integer directly to disc?

Alright, lets get this out of the way! The basics are pretty standard:

There are 8 bits in a byte

Bits either consist of a 0 or a 1

A byte can be interpreted in different ways, like binary octal or hexadecimal

Note: These are not character encodings, those come later. This is just a way to look at a set of 1’s and 0’s and see it in three different ways(or number systems).

Examples:

Input : 10011011

Output :

1001 1011 ---- 9B (in hex)

1001 1011 ---- 155 (in decimal)

1001 1011 ---- 233 (in octal)

This clearly shows a string of bits can be interpreted differently in different ways. We often use the hex representation of a byte instead of the binary one because it is shorter to write, this is just a representation and not an interpretation.

Text file in text editor (good)

Text file in hex editor (inconvenient)

By convention, the data in every text file obeys a number of rules:

The text looks readable to a human or at least moderately sane. Even if it contains a heavy proportion of punctuation symbols (like HTML, RTF, and other markup formats), there is some visible structure and it’s not seemingly random garbage.

The data format is usually line-oriented. Each line could be a separate command, or a list of values could put each item on a different line, etc. The maximum number of characters in each line is usually a reasonable value like 100, not like 1000.

Non-printable ASCII characters are discouraged or disallowed. Examples include the NUL byte (0x00), DEL byte (0x7F), and most of the range 0x01 to 0x1F (except tab, carriage return, newline, etc.). Some text editors silently convert or discard these bytes, which is why binary files should never be edited in a text editor.

The reading of [newline sequences](https://en.wikipedia.org/wiki/Newline) is usually universal – namely, CR (classic Mac OS), LF (Unix), or CR+LF (Windows) all mean the same thing, which is to end the current line and start the next one. The writing of newline sequences usually normalizes to the preferred one on the current platform, regardless of which variant was read. (For example, a text file “Hello CR LF world CR Lorem LF ipsum” read in a Unix text editor would likely be written out as “Hello LF world LF Lorem LF ipsum”.)

There is some [character encoding](https://en.wikipedia.org/wiki/Character_encoding) that governs how extended-ASCII bytes are handled. Byte values from 0x80 to 0xFF are not covered by the universally accepted ASCII standard, and the interpretation of these bytes depends on the choice of character encoding – such as UTF-8, ISO-8859-1, Shift JIS. Thus the interpretation of a text file depends on the character encoding used (unless the file format is known to be pure-ASCII), whereas a binary file is just a sequence of plain bytes with no inherent notion of character encoding.

In most text file formats, some flexibility is given to [whitespace](https://en.wikipedia.org/wiki/Whitespace_character) characters. For example, using one space or two space might not change the meaning of a command. And in C-like programming languages, one whitespace character has the same meaning as any positive number of whitespace or newline characters (except within strings).

Observations regarding the general computing environment around text files:

Any text file can be viewed or edited in any [text editor](https://en.wikipedia.org/wiki/Text_editor). The creator of a text-based file format doesn’t need to know or care about what editor program a future user will use. This contrasts with binary formats, where each format is generally coupled to a specific program that can handle it.

Software developers work with many kinds of text files all the time – program source code, setup scripts, technical documentation, program-generated output and logs, configuration files, you name it. Much of the world of computer programming revolves around text files, since they are easy to create, edit, and consume. This contrasts with how an average computer user mainly works with binary files.

Text files are usually viewed and edited in a [monospaced font](https://en.wikipedia.org/wiki/Monospaced_font). The historical reason is that the earliest computer terminals could only display monospaced text. A modern reason is that monospaced allows an author to visually align characters in different lines for [ASCII art](https://en.wikipedia.org/wiki/ASCII_art), tabular data, etc.

On the Unix platform, text files are so ubiquitous that they often have no file name extension at all. For example, “README” and “config” are the full names of popular files that exist in many places.

Beyond text editors, there exist many standard Unix tools to generate, manipulate, or view textual data: ls, more, sort, grep, awk, etc.

[Version control software](https://en.wikipedia.org/wiki/Version_control) (Git, Mercurial, SVN, Perforce, etc.) work best with text files. For binary files they can only tell you whether a file has changed or not. For text files, they can show line-by-line differences, perform automatic merges, and do many other useful things.

Encoding

Now that we know what a byte is and what it looks like, let us see how it is interpreted, mainly in strings. Character Encodings are a way to assign values to bytes or sets of bytes that represent a certain character in that scheme. Some encodings are ASCII(probably the oldest), Latin, and UTF-8(most widely used as of today. In a sense encodings are a way for computers to represent, send and interpret human readable characters. This means that a sentence in one encoding might become completely incomprehensible in another encoding.

Python and Bytes

From a developer’s point of view, the largest change in Python 3 is the handling of strings. In Python 2, the str type was used for two different kinds of values – text and bytes, whereas in Python 3, these are separate and incompatible types. This means that before Python3 we could treat a set of bytes as a string and work from there, this is not the case now, now we have a separate data type, called bytes. This data type can be briefly explained as a string of bytes, which essentially means, once the bytes data type is initialized it is immutable.

Example:

Python3

|  |
| --- |
| bytestr = bytes(b'abc')    # initializing a string with b  # makes it a binary string  print(bytestr)  print(bytestr[0])    bytestr[0] = 97 |

Output:

b'abc'

97

Traceback (most recent call last):

File "bytesExample.py", line 4, in

bytestr[0] = 97

TypeError: 'bytes' object does not support item assignment

A bytestring is what it says it is simply a string of bytes, for example ‘© ? ?’ in ‘utf-8’ is

b'\xC2\xA9\x20\xF0\x9D\x8C\x86\x20\xE2\x98\x83'

This presents another problem, we need to know the encoding of a binary string, because the same string in another encoding(latin-1) looks different.

Â© ð â

Q4. Describe a benefit of using the with keyword instead of explicitly opening a file.

The with keyword in Python is used as a [context manager](https://www.geeksforgeeks.org/context-manager-in-python/). As in any programming language, the usage of resources like file operations or database connections is very common. But these resources are limited in supply. Therefore, the main problem lies in making sure to release these resources after usage. If they are not released, then it will lead to resource leakage and may cause the system to either slow down or crash.

As we know, the [open()](https://www.geeksforgeeks.org/python-open-function/) function is generally used for file handling in [Python](https://www.geeksforgeeks.org/python-programming-language/). But it is a standard practice to use context managers like with keywords to handle files as it will automatically release files once its usage is complete.

Python with open() Syntax:

Syntax:

with open(file\_path, mode, encoding) as file:

        …

file\_path: It is the path to the file to open

mode: mode of operation on the file. ex.: read, write etc. (represented by r, w, r+, w+, rb, wb etc.)

encoding: read the file in correct encoding format.

Q5. Does Python have the trailing newline while reading a line of text? Does Python append a newline when you write a line of text?

Table of Contents - Python new line:

Python new line

Newline character in Python

Multi-line string

Closing thoughts - Python new line

Python new line:

In programming, it is a common practice to break lines and display content in a new line. This improves the readability of the output. Apart from that, you would frequently come across the new line character a lot while working with files.

Hence it is quite important that you understand how to add a new line and familiarise yourself with how the new line character works. This tutorial is aimed to do the same.

Newline character in Python:

In Python, the new line character “\n” is used to create a new line. When inserted in a string all the characters after the character are added to a new line. Essentially the occurrence of the “\n” indicates that the line ends here and the remaining characters would be displayed in a new line.

Code and Explanation:

str\_1 = "Hire the top \n1% freelance developers"

print(str\_1)

‘’’Output - Hire the top

1% freelance developers’’’

As aforementioned, the character after the new line character is printed in a new line.  
Different ways to implement this would include either adding it to string directly, or concatenating it before printing it. A common question that beginners have while learning how to apply a new line is - since we are adding it to a string - Why doesn’t Python print “\n” as it is? And how does it know that a new line must be added?

Well, the backslash (“\”) in the new line character is called an escape sequence. Escape sequences are used to add anything illegal to a string. This way Python understands that the following character is not a part of a string and executes it.

Multiline Strings:

Multiline strings are another easy way to print text in a new line. As the name suggests the string itself spans over multiple lines. These strings can be assigned by using either 3 double quotes or 3 single quotes. Python understands that the string is a multiline string and prints it as such.

Code and Explanation:

str\_1 = """Hire the top

1% freelance

developers"""

print(str\_1)

'''Output - Hire the top

1% freelance

developers'''

In the above example, the string is printed in the same way as the information was passed.

Closing thoughts - Python new line:

Although both methods can be used in Python to add new lines I would recommend using the first method as it is the most commonly accepted method. Also, given Python has an in-built character that facilitates this it is best to utilize it

Q6. What file operations enable for random-access operation?

RandomAccessFile

This [class](https://www.javatpoint.com/object-class)

is used for reading and writing to random access file. A random access file behaves like a large [array](https://www.javatpoint.com/array-in-java)

of bytes. There is a cursor implied to the array called file [pointer](https://www.javatpoint.com/c-pointers)

, by moving the cursor we do the read write operations. If end-of-file is reached before the desired number of byte has been read than EOFException is [thrown](https://www.javatpoint.com/throw-keyword)

. It is a type of IOException.

Random File Access in C

In the previous lessons, we learned how to open a file, close a file, read from a file, and write to a file. We also learned that there are two types of files: binary files and text files. Random file access means that you can take the file pointer to any part of the file for reading or writing. In general, with small files, we access the files sequentially. In sequential access, we access the file record by record or character by character. This approach is appropriate if the file size is small, but what if the file has tens of thousands of records? In random file access, if a file has 10,000 records and if you want to go to the 9,999th record, you can directly point the file pointer to this record using the random file access technique. This can be done using the functions we are about to mention.

[Save](https://study.com/academy/lesson/random-file-access-passing-filenames-returning-filenames-in-c-programming.html)

  Timeline

Autoplay

[Video](https://study.com/academy/lesson/random-file-access-passing-filenames-returning-filenames-in-c-programming.html#quiz-course-links)

[Quiz](https://study.com/academy/practice/quiz-worksheet-random-access-files-in-c.html)

[Course](https://study.com/academy/course/computer-science-111-programming-in-c.html)

20K views

fseek(), ftell(), rewind()

The fseek() function helps us send a file pointer to a specified location. The syntax of fseek() is as follows:

int fseek(FILE \*pointer, long int offset, int whence);

The function accepts three parameters; the file pointer, an integer offset (which is the number of bytes to be shifted from the position mentioned in the third parameter), and the third parameter (which specifies the position from where the offset is added). Along with this, there are three macros used in fseek(). These are SEEK\_SET (beginning of the file), SEEK\_CUR (current position in the file), SEEK\_END (end of the file).

The ftell() function is used to find out the exact position of the file pointer with respect to the beginning. This function accepts the file pointer as the parameter.

long ftell(FILE \*pointer);

The rewind() function also accepts the file pointer as a parameter and sets the file pointer to the beginning of the file again. The syntax is what you see appearing here:

void rewind(FILE \*pointer);

Example

Appearing here is an example using fseek(), ftell(), and rewind().

#include <stdio.h>  
int main () {   
  FILE \*fp;  
  int c;  
  fp = fopen("file.txt","w+");  
  fputs("This is study.com", fp);  
  // We are using fseek() to shift the file pointer to the 7th position.  
  fseek( fp, 7, SEEK\_SET );  
   
//Now we overwrite C programming in the 7th position  
  fputs(" C Programming", fp);  
   
//Now we print the current position of the file pointer using ftell()  
  printf("The current position of the file pointer is: %ld\n", ftell(fp));  
//We take the file pointer to the beginning of the file.  
  rewind(fp);  
   
//Now we verify if rewind() worked using ftell().  
  printf("The current position of the file pointer is: %ld\n", ftell(fp));  
  while(1) {   
   c = fgetc(fp);  
   if( feof(fp) ) {   
    break;  
   }  
   printf("%c", c);  
  }  
  fclose(fp);  
  return(0);  
}

Output

The current position of the file pointer is: 21  
The current position of the file pointer is: 0  
This is C Programming

Functions can be used to handle file operations only when you send the file pointer as a parameter to the function. You can also send the file name as a parameter and handle the operations inside the function. The common practice is to send the file pointer to a function for a specific purpose. This example has been modified to use a function for displaying the contents of the file by passing the file pointer.

Q7. When do you think you'll use the struct package the most?

The struct module in Python is used to convert native Python data types such as strings and numbers into a string of bytes and vice versa. What this means is that users can parse binary files of data stored in C structs in Python.

It is used mostly for handling binary data stored in files or from network connections, among other sources.

The module is only available in Python 3.x and needs to be imported first by writing

import struct

This process needs to be done at the start of the program.

Struct Functions

There are several functions built into the struct module. Some important ones are:

1. struct.pack()

struct.pack() is the function that converts a given list of values into their corresponding string representation. It requires the user to specify the format and order of the values that need to be converted.

The following code shows how to pack some given data into its binary form using the module’s struct.pack() function.

1

2

3

4

import struct

packed = struct.pack('i 4s f', 10, b'John', 2500)

print(packed)

The first argument of the function represents the format string. A \*format strings specifies the expected layout when packing and unpacking data. The rest of the arguments represent the data that needs to be packed.

These format strings are made up of format characters. Some common ones are:

2. struct.unpack()

This function converts the strings of binary representations to their original form according to the specified format. The return type of struct.unpack() is always a tuple.

1.import struct

2.packed = b'\n\x00\x00\x00John\x00@\x1cE'

3.unpacked = struct.unpack('i 4s f', packed)

4.print(unpacked)

The function is given a format string and the binary form of data. This function is used to parse the binary form of data stored as a C structure.

3. struct.calcsize()

This function calculates the size of the String representation of struct with a given format string

1.import·struct

2.size·=·struct.calcsize('i·4s·f')

3.print("Size·in·bytes:·{}".format(size))

Q8. When is pickling the best option?

Pickle in Python is primarily used in serializing and deserializing a Python object structure. In other words, it's the process of converting a Python object into a byte stream to store it in a file/database, maintain program state across sessions, or transport data over the network.

In Python, you can use pickle to serialize (deserialize) an object structure into (from) a byte stream. Here are best practices for secure Python pickling.

By [Ashutosh Agrawal](https://www.synopsys.com/blogs/software-security/author/aagrawal/), senior consultant, and [Arvind Balaji](https://www.synopsys.com/blogs/software-security/author/abalaji/), associate consultant, Synopsys

Pickle in Python is primarily used in [serializing and deserializing a Python object structure](https://wiki.python.org/moin/UsingPickle). In other words, it’s the process of converting a Python object into a byte stream to store it in a file/database, maintain program state across sessions, or transport data over the network. The pickled byte stream can be used to re-create the original object hierarchy by unpickling the stream. This whole process is similar to object serialization in Java or .Net.

When a byte stream is unpickled, the pickle module creates an instance of the original object first and then populates the instance with the correct data. To achieve this, the byte stream contains only the data specific to the original object instance. But having just the data alone may not be sufficient. To successfully unpickle the object, the pickled byte stream contains instructions to the unpickler to reconstruct the original object structure along with instruction operands, which help in populating the object structure.

According to the pickle module documentation, the following types can be pickled:

None, true, and false

Integers, long integers, floating point numbers, complex numbers

Normal and Unicode strings

Tuples, lists, sets, and dictionaries containing only picklable objects

Functions defined at the top level of a module

Built-in functions defined at the top level of a module

Classes that are defined at the top level of a module

Pickle allows different objects to declare how they should be pickled using the \_\_reduce\_\_ method. Whenever an object is pickled, the \_\_reduce\_\_ method defined by it gets called. This method returns either a string, which may represent the name of a Python global, or a tuple describing how to reconstruct this object when unpickling.

Generally the tuple consists of two arguments:

A callable (which in most cases would be the name of the class to call)

Arguments to be passed to the above callable

The pickle library will pickle each component of the tuple separately, and will call the callable on the provided arguments to construct the new object during the process of unpickling

Q9. When will it be best to use the shelve package?

A “shelf” is a persistent, dictionary-like object. The difference with “dbm” databases is that the values (not the keys!) in a shelf can be essentially arbitrary Python objects — anything that the [pickle](https://docs.python.org/3/library/pickle.html#module-pickle) module can handle. This includes most class instances, recursive data types, and objects containing lots of shared sub-objects. The keys are ordinary strings.

shelve.open(filename, flag='c', protocol=None, writeback=False)

Open a persistent dictionary. The filename specified is the base filename for the underlying database. As a side-effect, an extension may be added to the filename and more than one file may be created. By default, the underlying database file is opened for reading and writing. The optional flag parameter has the same interpretation as the flag parameter of [dbm.open()](https://docs.python.org/3/library/dbm.html" \l "dbm.open" \o "dbm.open).

By default, pickles created with [pickle.DEFAULT\_PROTOCOL](https://docs.python.org/3/library/pickle.html" \l "pickle.DEFAULT_PROTOCOL" \o "pickle.DEFAULT_PROTOCOL) are used to serialize values. The version of the pickle protocol can be specified with the protocol parameter.

Because of Python semantics, a shelf cannot know when a mutable persistent-dictionary entry is modified. By default modified objects are written only when assigned to the shelf (see [Example](https://docs.python.org/3/library/shelve.html#shelve-example)). If the optional writeback parameter is set to True, all entries accessed are also cached in memory, and written back on [sync()](https://docs.python.org/3/library/shelve.html#shelve.Shelf.sync) and [close()](https://docs.python.org/3/library/shelve.html#shelve.Shelf.close); this can make it handier to mutate mutable entries in the persistent dictionary, but, if many entries are accessed, it can consume vast amounts of memory for the cache, and it can make the close operation very slow since all accessed entries are written back (there is no way to determine which accessed entries are mutable, nor which ones were actually mutated).

Changed in version 3.10: [pickle.DEFAULT\_PROTOCOL](https://docs.python.org/3/library/pickle.html" \l "pickle.DEFAULT_PROTOCOL" \o "pickle.DEFAULT_PROTOCOL) is now used as the default pickle protocol.

Changed in version 3.11: Accepts [path-like object](https://docs.python.org/3/glossary.html#term-path-like-object) for filename.

Q10. What is a special restriction when using the shelve package, as opposed to using other data dictionaries?

The shelve module in Python’s standard library provides simple yet effective object persistence mechanism. The shelf object defined in this module is dictionary-like object which is persistently stored in a disk file. This creates a file similar to dbm database on UNIX like systems.

The shelf dictionary has certain restrictions. Only string data type can be used as key in this special dictionary object, whereas any picklable Python object can be used as value.

The shelve module defines three classes as follows −

|  |  |
| --- | --- |
| Sr.No | Shelve Module & Description |
| 1 | Shelf  This is the base class for shelf implementations. It is initialized with dict-like object. |
| 2 | BsdDbShelf  This is a subclass of Shelf class. The dict object passed to its constructor must support first(), next(), previous(), last() and set\_location() methods. |
| 3 | DbfilenameShelf  This is also a subclass of Shelf but accepts a filename as parameter to its constructor rather than dict object. |

The open() function defined in shelve module which return a DbfilenameShelf object.

open(filename, flag='c', protocol=None, writeback=False)

The filename parameter is assigned to the database created. Default value for flag parameter is ‘c’ for read/write access. Other flags are ‘w’ (write only) ‘r’ (read only) and ‘n’ (new with read/write).

The serialization itself is governed by pickle protocol, default is none. Last parameter writeback parameter by default is false. If set to true, the accessed entries are cached. Every access calls sync() and close() operations, hence process may be slow.

Following code creates a database and stores dictionary entries in it.

import shelve

s=shelve.open("test")

s['name']="Ajay"

s['age']=23

s['marks']=75

s.close()

This will create test.dir file in current directory and store key-value data in hashed form. The Shelf object has following methods available −

|  |  |
| --- | --- |
| Sr.No. | Methods & Description |
| 1 | close()  synchronise and close persistent dict object. |
| 2 | sync()  Write back all entries in the cache if shelf was opened with writeback set to True. |
| 3 | get()  returns value associated with key |
| 4 | items()  list of tuples – each tuple is key value pair |
| 5 | keys()  list of shelf keys |
| 6 | pop()  remove specified key and return the corresponding value. |
| 7 | update()  Update shelf from another dict/iterable |
| 8 | values()  list of shelf values |

To access value of a particular key in shelf −

s=shelve.open('test')

print (s['age']) #this will print 23

s['age']=25

print (s.get('age')) #this will print 25

s.pop('marks') #this will remove corresponding k-v pair

As in a built-in dictionary object, the items(), keys() and values() methods return view objects.

print (list(s.items()))

[('name', 'Ajay'), ('age', 25), ('marks', 75)]

print (list(s.keys()))

['name', 'age', 'marks']

print (list(s.values()))

['Ajay', 25, 75]

To merge items of another dictionary with shelf use update() method.

d={'salary':10000, 'designation':'manager'}

s.update(d)

print (list(s.items()))

[('name', 'Ajay'), ('age', 25), ('salary', 10000), ('designation', 'manager')]