00010000

CHAPTER 16

File I/O

Learning Objectives

- *State the purpose of a file*
- *Use the open() function to create a file object*
- *Use a context manager when conducting file I/O operations*
- *List and describe the text file modes*
- *List and describe the binary file modes*
- *List and describe the file object methods*
- *Write strings to a file with the write() method*
- *Explain the difference between the read() and readline() methods*
- *Demonstrate your ability to interact with text and binary files*
- *State the purpose of a file pointer*
- *Write JSON data to a file*
- *Read JSON data from a file*
- *Explain the purpose of UTF-8 encoding*
- *Conduct random file I/O operations*

Introduction

You will often need to store data in a file on disk for later use, either to preserve application state between runs or to store the results of processing during program execution. The types of data you might want to preserve include plain text such as JSON, XML, and comma separated values (CSV), binary data such as images in JPEG, PNG, or TIFF format, or serialized objects created with the Python pickle module, to name a few.

Luckily, Python makes it easy to work with files. In this chapter, you'll learn how to open a file for read, write, append, and update operations. You'll learn about absolute and relative file paths. You'll learn how to save JSON data to a file. You'll also learn the difference between binary files and text files along with their associated file modes.

An important question to ask yourself when saving data to a file is how you intend to share the data. Plain text files that contain data in JSON, XML, or CSV format are easiest to share because they can be read and processed by any programming language.

Note that in this chapter I focus solely on file I/O, but many of the concepts you learn here apply equally well to other forms of I/O including memory streams and network socket streams.

1 Basic File Operations

The beautiful thing about Python is that it lets you to do a lot with just a little bit of code. Let's start by writing and reading data to and from a text file as shown in example 16.1.

16.1 basic_file_ops.py

```python
"""Demonstrate basic file operations."""


def main():
    filename = 'data.txt'
    input_text = None

    try:
        while True:
            # Get user's input
            input_text = input('Enter some text: ')

            # Exit program if user enters 'quit'
            if input_text == 'quit':
                exit()

            # Write user's input to a file
            with open(filename, 'w') as f:
                f.write(input_text)

            input('Press any key to continue: ')

            # read text from a file
            with open(filename, 'r') as f:
                print(f'You Entered: {f.read()}')

    except (OSError, Exception) as e:
        print(f'Problem writing file: {e}')
```

```
29
30
31   if __name__ == '__main__':
32       main()
33
```

Referring to example 16.1 — There's a lot going on in this short program. Overall, the program runs a continuous `while` loop that prompts the user for input. If the user enters `'quit'` the program exits, otherwise, the program opens a file named `data.txt` in *write text* mode and writes the user's input to the file. The program then asks the user to press any key to continue, opens the `data.txt` file in *read text* mode, reads the file contents, and prints it to the console.

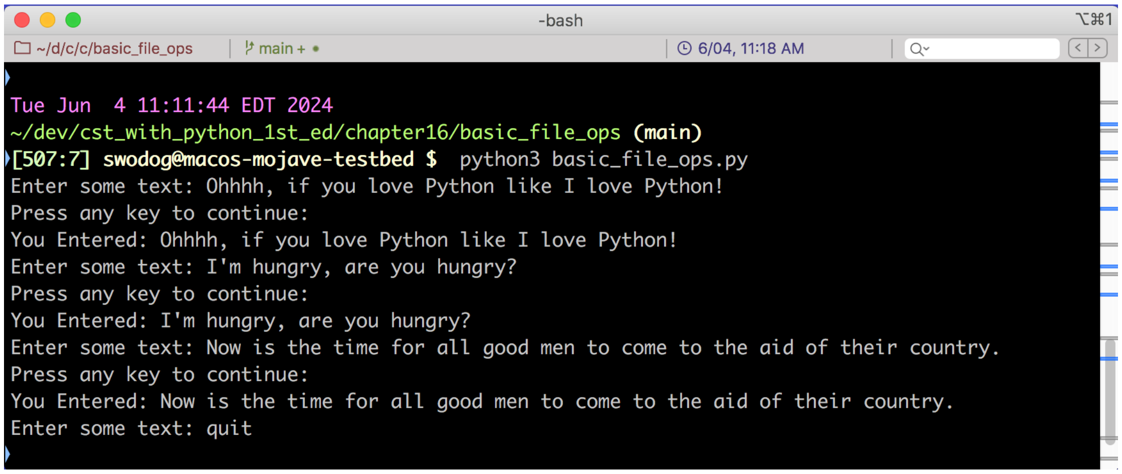
Let's break down further the code that opens and writes to the file. First, however, note that all the important code is enclosed within a `try/except` statement. This is necessary because a lot can go wrong when conducting file I/O operations. Notice as well on line 27 that I'm handling two types of exceptions: `OSError` and `Exception`. This is an example of how multiple exceptions must be packaged as tuples.

Looking at lines 18 and 19, I'm using the context manager `with` keyword along with the built-in function `open()` to open a file named `data.txt` for writing. The first argument to the `open()` function is the *name* of the file to open; the second argument is the *file mode*. In this case, I want to write to the file so I'm using the `'w'` file mode. The `'w'` file mode is destructive in that it will overwrite an existing file with the same name. More on file modes later.

The '`as f`' assigns an *alias* to the file object. The letter `'f'` is idiomatic, meaning it has become an accepted name for a file object alias. You are free to use any alias you want. On line 19, the call to `f.read()` reads the file's contents in its entirety.

The purpose of the `with` context manager is to automatically manage the file resource. If you didn't use the context manager, you would need to explicitly `close()` the file when you finish the `read()` operation or in case of an error. The context manager allows you to forget about micro managing file resources and I'll be using it in this chapter and throughout the book.

When lines 18 and 19 complete execution, the user's input has been written to a file named `data.txt` in the *local* or *working* directory. The working directory is the directory where this example program executes. Lines 24 and 25 open the `data.txt` file for reading and display its contents in the console. Figure 16-1 shows the results of running this program with various input.



```
Tue Jun  4 11:11:44 EDT 2024
~/dev/cst_with_python_1st_ed/chapter16/basic_file_ops (main)
[507:7] swodog@macos-mojave-testbed $  python3 basic_file_ops.py
Enter some text: Ohhhh, if you love Python like I love Python!
Press any key to continue:
You Entered: Ohhhh, if you love Python like I love Python!
Enter some text: I'm hungry, are you hungry?
Press any key to continue:
You Entered: I'm hungry, are you hungry?
Enter some text: Now is the time for all good men to come to the aid of their country.
Press any key to continue:
You Entered: Now is the time for all good men to come to the aid of their country.
Enter some text: quit
```

Figure 16-1: Results of Running Example 16.1 with Various Inputs (Your Results May Vary)

Referring to figure 16-1 — Each loop through the program overwrites the `data.txt` file. The last input string remains in the file. Example 16.2 shows the contents of the `data.txt` file after the program exists. Your results will most certainly be different unless you enter the same text as shown in figure 16-1.

16.2 data.txt

```
Now is the time for all good men to come to the aid of their country.
```

1.1 File Modes

The complete signature of the built-in `open()` function is shown in the following code snippet:

```
open(file, mode='r', buffering=-1, encoding=None, errors=None,
    newline=None, closefd=True, opener=None)
```

Of the eight parameters, you will most often use the first two: *file* and *mode*. You may need to set the remaining parameters depending on the type of data and size of the file.

A file is opened in a particular mode given by the second argument to the `open()` function. Python treats files as containing either *text* or *binary* data. The default file mode is `'r'` for *read text* (`'r'` is a synonym for `'rt'`), which returns the content of the file as a string (`str`) in the default platform encoding or the encoding given by the `open()` function's `encoding` parameter. To read a file as binary data you would use the file mode `'rb'` for *read binary*. Table 16-1 lists the various file modes and their meaning.

File Mode	Meaning
`'r'`	Open file for reading text. This is the default file mode. Returns file contents as a string (`str`). File pointer is set to the beginning of the file.
`'w'`	Open file for writing. Overwrites the file if it already exists. Creates new file if it does not exist. Sets file pointer to beginning of file.
`'x'`	Open file for exclusive creation. Fails if the file already exists. Creates file if it does not exist. Sets file pointer to beginning of file.
`'a'`	Open file for appending. Appends data to the end of the file if it already exists. File pointer set to end of file.
`'b'`	Open file in binary mode. Returns file contents as a binary object.
`'t'`	Open file in text mode. This is the default file mode. (i.e., `'r'` is the same as `'rt'`)
`'+'`	Open the file for updating. (i.e., reading and writing)

Table 16-1: File Modes and Their Meaning

Referring to table 16-1 — As you can see, the default file mode is `'r'` which stands for `'read text'`. Text mode `'t'` is inferred if binary mode `'b'` is not specified. In other words, the file mode `'w'` stands for `'write text'`, file mode `'a'` stands for `'append text'`, etc.

The *file pointer* is a property associated with a file object that indicates where the next read or write will occur within the file. You'll learn more about file pointers in the section *Binary Data and Random File I/O* later in this chapter.

File modes `'r'`, `'w'`, `'a'`, and `'x'` can be used in combination with the binary `'b'` and update `'+'` modes as shown in table 16-2.

File Mode	Meaning
`'r+'`	Open file for both reading and writing text. File pointer is set to beginning of file.
`'rb'`	Open the file for reading binary data. File pointer is set to beginning of file.
`'rb+'`	Open file for reading and writing binary data. File pointer is set to beginning of file.
`'w+'`	Open file for writing and reading text. Overwrites file if it exists. Creates a new file if it does not exist.
`'wb'`	Open file for writing binary data. Overwrites file if it exists. Creates a new file if it does not.
`'wb+'`	Open file for writing and reading binary data. Overwrites file if it exists. Creates a new file if it does not.
`'a+'`	Open file for appending and reading text. Creates a new file if it does not exist. Sets file pointer to end of file.
`'ab'`	Open file for appending binary data. Creates new file if it does not exist. Sets file pointer to end of file.
`'ab+'`	Opens file for appending and reading binary data. Creates new file if it does not exist. Sets file pointer to end of file.
`'xb'`	Open file for binary data writing if it does not already exist.
`'xb+'`	Open file for binary data updating if it does not already exist.

Table 16-2: File Mode Combinations

1.2 The File Object And Its Methods

Calling the `open()` function creates and returns a *file object*. A file object provides a set of methods you can use to interact with a file on disk. The methods available on a file object depend on whether you are dealing with text data, binary data, or random access file operations. For example, opening a file with file mode `'r+'` will create a file object that expects string input and returns string output. Attempting to send binary data to a text-mode file object will raise a `TypeError` exception. Table 16-3 lists the methods exposed by file objects and their meaning.

Method	Meaning
Frequently-Used Methods	

`f.read(size:int=-1)->str`
`f.read(size:int=-1)->bytes`
	Read number of characters or bytes from file up to size. If size not provided, reads the contents of the file in its entirety. Returns string if file opened in text mode and bytes if opened in binary mode.

Table 16-3: File Object Methods with Type Hints

Method	Meaning
`f.readline(size:int=-1)->str`	Reads a single line of text from file. A newline character `'\n'` remains on each line with the exception of the last line of text. Thus, if `readline()` returns an empty string it indicates the end of the file. If it returns only a newline character it indicates a blank line. If `size` is provided, reads up to that many bytes from the line.
`f.readlines(hint:int=-1)->list[str]`	Read all the lines in a text file and return a list of strings. If `hint` is provided, reads line until total size in bytes reaches `hint`.

`f.write(data:str)->int`
`f.write(data:bytes)->int`
`f.write(data:bytearray)->int`
	Write data to file and return number of characters or bytes written.

`f.writelines(lines:list[str])->int`	Writes a list of `lines` to the file. Does not add a newline character between lines.
`f.seek(offset:int, whence:int=0)->int`	Move file pointer to indicated `offset` from reference point (`whence`). The `whence` argument is optional and defaults to 0 indicating from the beginning of the file. Values for `whence` can be `0` (beginning of file), `1` (current position), and `2` (end of file)
`tell()->int`	Returns current file pointer position.
Infrequently-Used Methods	
`f.close()->None`	Close the file. Context manager (`with`) will close the file automatically.
`f.flush()->None`	Flush the file stream write buffer. Has no effect on read-only files.
`f.fileno()->int`	Returns a file's low-level file descriptor number if it exists.
`f.isatty()->bool`	Returns True if the underlying I/O stream is a terminal.
`f.truncate(size:int=None)`	Resize the file to `size` in bytes. Can be used to reduce or extend the file size. Extended file space is usually filled with zero-valued bytes.
`f.seekable()->bool`	Returns `True` if the file supports random access.
`f.readable()->bool`	Returns `True` if the file is opened in a read or update mode.
`f.writable()->bool`	Returns `True` if the file is opened in write, append, or update

Table 16-3: File Object Methods with Type Hints (Continued)

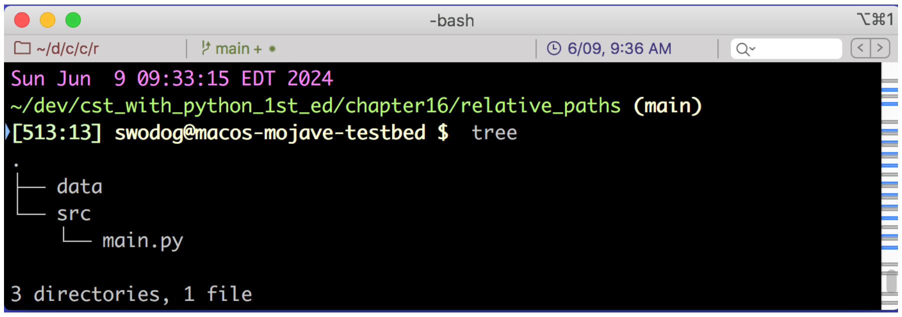
Referring to table 16-3 — I've grouped the file object methods into two groups representing how often you are likely to use each method. I remind you once again that the methods available on a file object and the data types they support depend on the mode used to open the file. For the infrequently-used methods, opening a file with a context manager (i.e., using the `with` keyword) eliminates the need to call the `close()` method. Since you are the programmer, if you open a file in `'r'` mode, you know it's readable and not writable, so there's generally no need to check for these capabilities. I've never encountered the need to use the `truncate()` method, but that's just me.

1.3 File Paths

A critical skill you need to cultivate is how to navigate your computer's file system. Referring again to example 16.1 — I provide only the file name and mode to the `open()` method. The filename provides no explicit path information, so the file is opened in the *working directory*. The working directory defaults to the directory in which the `python` command executes. It's best to avoid writing files to the working directory. Instead, you should write program data files to a dedicated directory. To read and write files from different directories you need to understand file paths, referred to simply as *paths*, and the differences between *relative* and *absolute* paths.

1.3.1 Relative Paths

Relative paths are formulated from the starting point of a particular directory. Let's say I am working on a project and have my Python source files in a `src` directory and I want to store files in a dedicated directory named `data` located in the project directory. Figure 16-2 shows how this might look.



```
Sun Jun  9 09:33:15 EDT 2024
~/dev/cst_with_python_1st_ed/chapter16/relative_paths (main)
[513:13] swodog@macos-mojave-testbed $  tree
.
├── data
└── src
    └── main.py

3 directories, 1 file
```

Figure 16-2: Project Directory Layout with `src` and `data` Directories

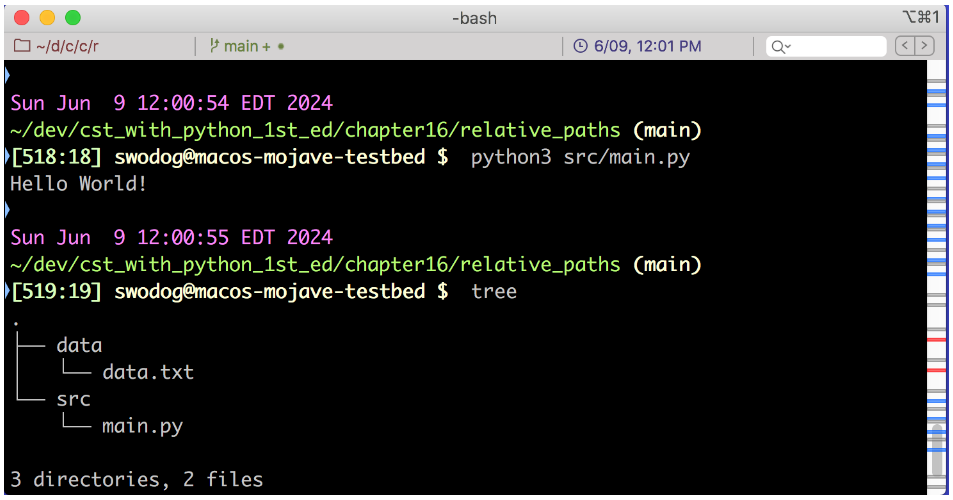
Referring to figure 16-2 — The name of the project directory is `relative_paths`. The full or absolute path to the project directory is: `~/dev/cst_with_python_1st_ed/chapter16/relative_paths` on macOS (Unix). The `relative_paths` directory contains two subdirectories: `src` and `data`. The `src` directory contains one source file: `main.py`. Example 16.3 lists main.py.

16.3 main.py

```
"""Demonstrate writing files to relative paths."""

def main():
    # Bad practice -- Don't do this!
    file_name = 'data/data.txt'

    try:
        with open(file_name, 'w') as f:
            f.write('Hello World!')

        with open(file_name, 'r') as f:
            print(f'{f.read()}')

```

```
14      except (OSError, Exception) as e:
15          print(f'Problem writing file: {e}')
16
17
18  if __name__ == '__main__':
19      main()
20
```

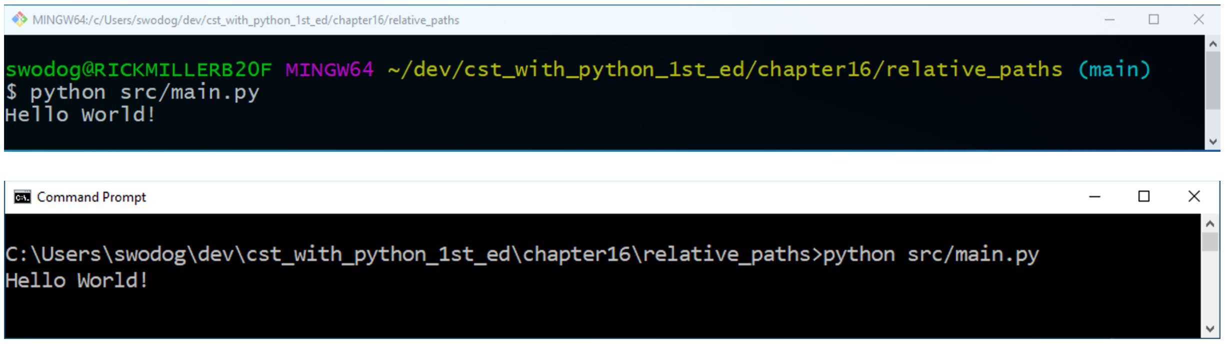
Referring to example 16.3 — On line 4 I've hard coded a relative file path which includes the name of the `data` directory, a forward slash, which is the Unix path separator `'/'`, and the file name `data.txt`. This program may or may not work correctly depending on where I execute the `python` command. If I am currently in the project directory, `relative_paths`, and run `main.py` from the command line like so...`python3 src/main.py`...then the working directory is set to `relative_paths` and the program successfully locates the `data` directory. Figure 16-3 shows the results of running the program from the `relative_paths` directory.



```
Sun Jun  9 12:00:54 EDT 2024
~/dev/cst_with_python_1st_ed/chapter16/relative_paths (main)
[518:18] swodog@macos-mojave-testbed $  python3 src/main.py
Hello World!

Sun Jun  9 12:00:55 EDT 2024
~/dev/cst_with_python_1st_ed/chapter16/relative_paths (main)
[519:19] swodog@macos-mojave-testbed $  tree
.
├── data
│   └── data.txt
└── src
    └── main.py

3 directories, 2 files
```

Figure 16-3: Results of Running Example 16.3

Referring to figure 16-3 — When I execute `main.py` in the project directory, the program can find the relative path to the `data` directory. However, another problem can potentially arise because I hardcoded the path with the Unix path separator. What will happen if I run this program on a Windows machine. Figure 16-4 shows the results.



```
swodog@RICKMILLERB20F MINGW64 ~/dev/cst_with_python_1st_ed/chapter16/relative_paths (main)
$ python src/main.py
Hello World!
```

```
C:\Users\swodog\dev\cst_with_python_1st_ed\chapter16\relative_paths>python src/main.py
Hello World!
```

Figure 16-4: Running Example 16.3 in Windows Git Bash (top) and Command Prompt Terminals (above)

Referring to figure 16-4 — It appears I got lucky. Since I developed example 16.3 on macOS, a Unix certified operating system, the program works as expected with the forward slash file separator. When I test it on Windows in Git Bash it also works as expected, and it also works fine in the Windows Command Prompt, which actually surprised me. It seems modern Windows operating systems know how to handle Unix file path separators. However, if I were a Windows developer and used a backslash vs. a forward slash in the hard coded file path, things would run fine in Windows, but would break on macOS and Linux. This leads to the following Pro Tip:

Pro Tip: Do not hard code file separators in path strings.

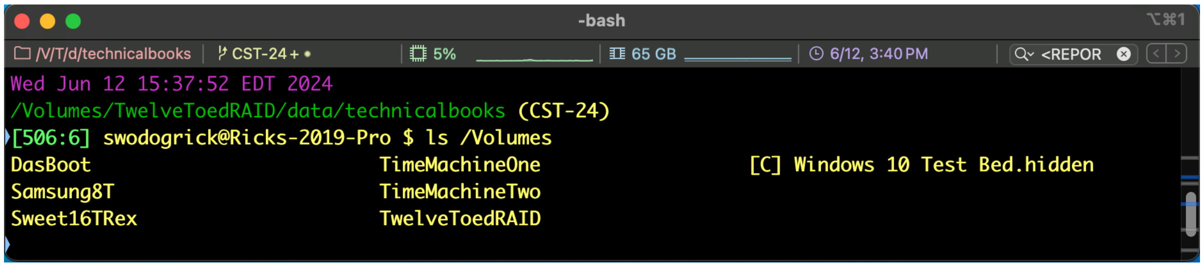
Later, in the section *Forming OS-Agnostic File Paths*, I'll show you how to use the `os` packages's `os.path.join()` method to form file paths that work equally well across all three operating systems macOS, Linux, and Windows.

1.3.2 Absolute Paths

An absolute path is a fully-qualified path that starts at the root of the file system.

1.3.2.1 Linux and macOS

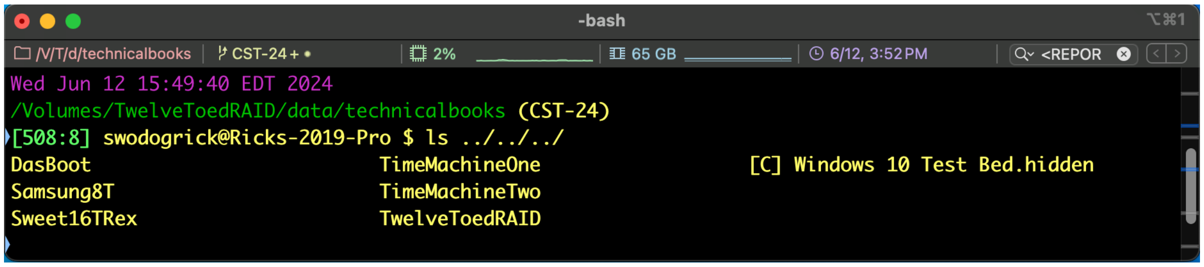
On macOS and Linux the file system root begins with a forward slash `'/'`. Hard drives, SSDs, DVDs, thumb drives, etc, connected to macOS are located in the `'/Volumes'` directory. On Linux they can be found in `'/dev'` or `'/mnt'`. Figure 16-5 shows the devices mounted under my `'/Volumes'` directory.



```
Wed Jun 12 15:37:52 EDT 2024
/Volumes/TwelveToedRAID/data/technicalbooks (CST-24)
[506:6] swodogrick@Ricks-2019-Pro $ ls /Volumes
DasBoot                         TimeMachineOne                  [C] Windows 10 Test Bed.hidden
Samsung8T                       TimeMachineTwo
Sweet16TRex                     TwelveToedRAID
```

Figure 16-5: Devices Mounted Under `'/Volumes'` Directory on My Mac Pro

Referring to figure 16-5 — Notice that I'm listing the drives in the `'/Volumes'` directory from the `'/Volumes/TwelveToedRAID/data/technicalbooks'` directory. Beginning a path with a forward slash on macOS and Linux indicates it is an absolute path starting from the root directory. Alternatively, I could use a relative path to achieve the same results as shown in figure 16-6.



```
Wed Jun 12 15:49:40 EDT 2024
/Volumes/TwelveToedRAID/data/technicalbooks (CST-24)
[508:8] swodogrick@Ricks-2019-Pro $ ls ../../../
DasBoot                         TimeMachineOne                  [C] Windows 10 Test Bed.hidden
Samsung8T                       TimeMachineTwo
Sweet16TRex                     TwelveToedRAID
```

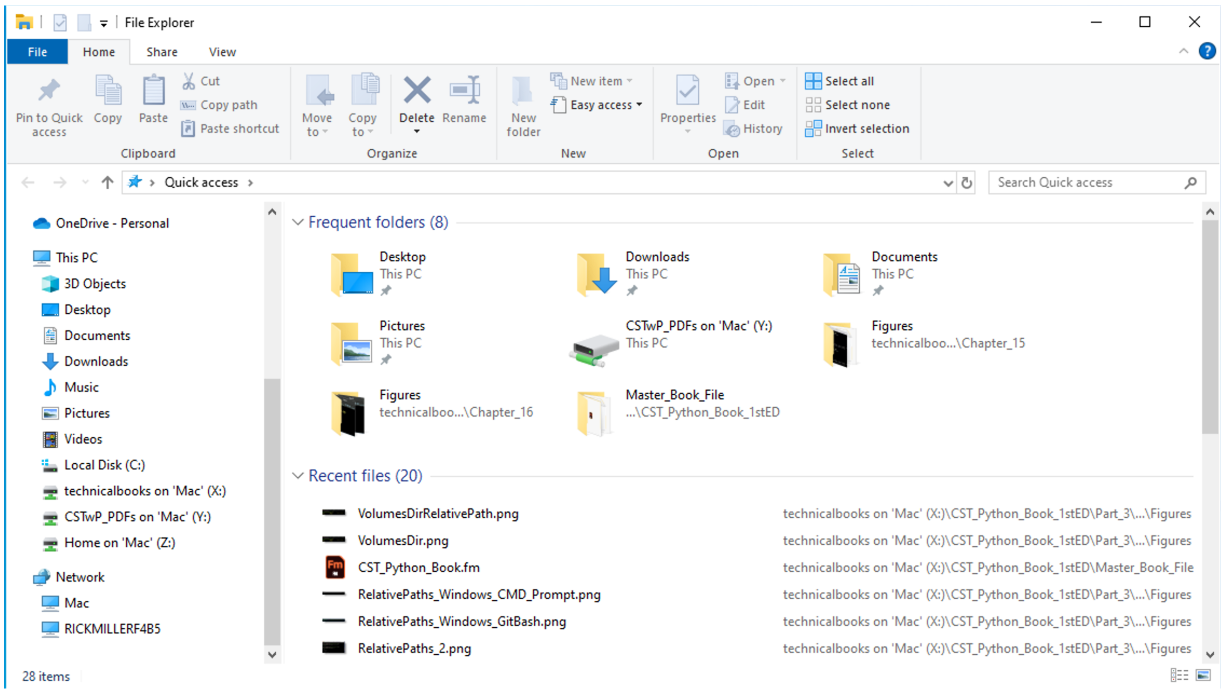
Figure 16-6: Accessing `'/Volumes'` via a Relative Path

Referring to figure 16-6 — Notice here that I'm using `'../../../'` to indicate three directories up from the current directory.

1.3.2.2 Windows

Windows file systems begin with a drive letter. Back in the day, yes, perhaps I dare say, before you were born, drive letters 'A' and 'B' indicated floppy drives, while 'C' was assigned to the hard drive, if you were lucky enough to have two hard drives, the second one was assigned the letter 'D'. I'm referring to the days before Windows but I am dating myself. I haven't seen a floppy drive in ions.

An absolute path on a Windows machine starts with a drive letter. Figure 16-7 shows some of the drives and network folders attached to the Windows 10 VM I am currently using to write this book.



VolumesDirRelativePath.png	technicalbooks on 'Mac' (X:)\CST_Python_Book_1stED\Part_3\...\Figures
VolumesDir.png	technicalbooks on 'Mac' (X:)\CST_Python_Book_1stED\Part_3\...\Figures
CST_Python_Book.fm	technicalbooks on 'Mac' (X:)\CST_Python_Book_1stED\Master_Book_File
RelativePaths_Windows_CMD_Prompt.png	technicalbooks on 'Mac' (X:)\CST_Python_Book_1stED\Part_3\...\Figures
RelativePaths_Windows_GitBash.png	technicalbooks on 'Mac' (X:)\CST_Python_Book_1stED\Part_3\...\Figures
RelativePaths_2.png	technicalbooks on 'Mac' (X:)\CST_Python_Book_1stED\Part_3\...\Figures

Figure 16-7: Windows Drives Have Assigned Letters

Referring to figure 16-7 — The Local Disk (C:) is the `'C'` drive. The network connected folders are assigned the letters `'X'`, `'Y'`, and `'Z'`. If I insert a thumb drive, Windows will automatically assign it a letter. To formulate an absolute file path on a Windows machine start with a volume's letter, as, for example, the path to the Program Files directory on the `'C'` drive `"C:\Program Files"`.

1.3.3 Forming OS-Agnostic File Paths

Repeating once again, **do not** hard code paths in your Python code. The preferred way to formulate paths is to use the os package's `os.path.join()` method. You can also check to ensure a directory exists before writing files to it as shown in example 16.4.

16.4 os_agnostic_paths.py

```python
"""Demonstrate the built-in os package."""

import os

def main():
    working_dir = os.getcwd()
```

```
7       data_dir = 'data'
8       data_dir_path = os.path.join(working_dir, data_dir)
9       file_name = 'data.txt'
10
11      print(f'Working Directory: {working_dir}')
12      print(f'Data Directory: {data_dir_path}')
13
14      if not os.path.exists(data_dir_path):
15         os.makedirs(data_dir_path)
16
17      try:
18         with open(os.path.join(data_dir_path, file_name), 'w+') as f:
19             input_string = ''
20             while input_string != 'quit':
21                input_string = input('Enter a string or "quit" to exit: ')
22                if input_string != 'quit':
23                   f.write(f'{input_string}\n' )
24
25             f.seek(0)
26             print(f'{"*" * 20} File Contents {"*" * 20}')
27             print(f'{f.read()}')
28
29      except (OSError, Exception) as e:
30         print(f'Error! {e}')
31
32
33   if __name__ == '__main__':
34      main()
35
```

Referring to example 16.4 — This program properly formulates file paths on all three operating systems: Linux, macOS, and Windows. Starting on line 3, I import the `os` package. On line 6, I create a variable named `working_dir` with the help of the `os.getcwd()` method, which returns the absolute path to the current working directory. On line 8, I create the variable named `data_dir_path` with the help of the `os.path.join()` method, which joins the `working_dir` and `data_dir` variables to form the absolute path to the data directory. On lines 11 and 12 I print the `working_dir` and `data_dir_path` values to the console. Next, in the `if` statement that begins on line 14, I check to see if the data directory already exists with the help of the `os.path.exists()` method and if not I create it with the help of the `os.makedirs()` method.

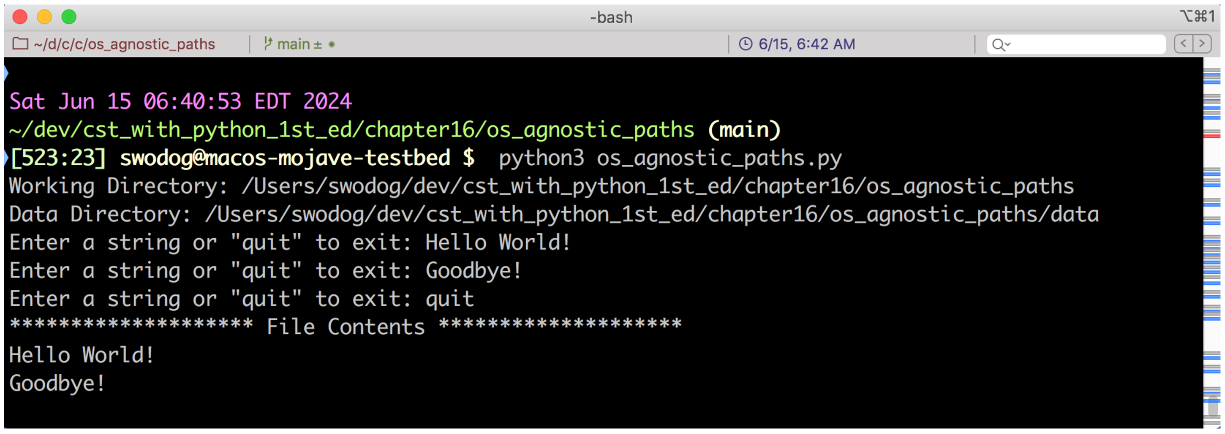
The meat of the program resides within the `try/except` block. I open the `data.txt` file for writing and updating and `while` the user enters anything besides `'quit'` the program writes the user's input to the file. When the user enters `'quit'`, the `while` loop exits, and on line 25 I reset the file pointer to the beginning of the file with the help of the `f.seek()` method. I then read the file and print its contents to the console.

Note on line 23 that I am adding a line feed `'\n'` character to the end of each line I write to the file.

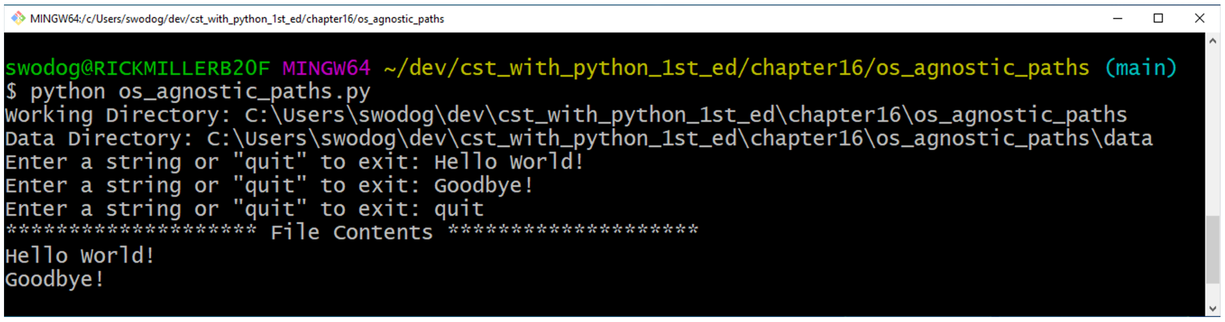
The use of `f.seek(0)` is required in this case because I am writing and reading the file within one context manager session. Each file write operation, at least in this example, leaves the file pointer set to the end of the file, which is were the next read or write operation will occur. If you perform a read operation with the file pointer set to the end of the file you will find nothing there. Remember, if you find yourself scratching your head wondering why you can see text in your file when you open it in a text editor but a read operation returns nothing...you need to reset the file pointer to the beginning of the file...and then read it.

Later, in the section on Random File I/O, you will learn how to move the file pointer to a specific location and read a specific number of bytes.

Figure 16-8 shows the results of running this program on macOS and Windows Git Bash.



```
Sat Jun 15 06:40:53 EDT 2024
~/dev/cst_with_python_1st_ed/chapter16/os_agnostic_paths (main)
[523:23] swodog@macos-mojave-testbed $  python3 os_agnostic_paths.py
Working Directory: /Users/swodog/dev/cst_with_python_1st_ed/chapter16/os_agnostic_paths
Data Directory: /Users/swodog/dev/cst_with_python_1st_ed/chapter16/os_agnostic_paths/data
Enter a string or "quit" to exit: Hello World!
Enter a string or "quit" to exit: Goodbye!
Enter a string or "quit" to exit: quit
******************** File Contents ********************
Hello World!
Goodbye!
```



```
swodog@RICKMILLERB20F MINGW64 ~/dev/cst_with_python_1st_ed/chapter16/os_agnostic_paths (main)
$ python os_agnostic_paths.py
Working Directory: C:\Users\swodog\dev\cst_with_python_1st_ed\chapter16\os_agnostic_paths
Data Directory: C:\Users\swodog\dev\cst_with_python_1st_ed\chapter16\os_agnostic_paths\data
Enter a string or "quit" to exit: Hello World!
Enter a string or "quit" to exit: Goodbye!
Enter a string or "quit" to exit: quit
******************** File Contents ********************
Hello World!
Goodbye!
```

Figure 16-8: Results of Running Example 16.4 on macOS (top) and Windows Git Bash (above)

Referring to figure 16-8 — Notice how the absolute paths are rendered in each operating system.

1.3.3.1 Parting Thoughts

The `os` package packs a heavy punch and makes it easy to formulate and work with OS-agnostic file paths. It lets you focus on the problem at hand and leave the operating system details to Python. Also, another great piece of advice, especially when working with files and file paths, if you plan to run your Python programs on different operating systems then you must test your program accordingly which leads to following Pro Tip:

Pro Tip: Make no assumptions about Python cross-platform compatibility. Use the os package to formulate and work with OS-agnostic file paths and test the hell out of your Python programs on every operating system you intend to run them on.

Quick Review

Python makes it easy to work with files. You can do a lot with just a little bit of code. Use the built-in `open( )` function to open a file in a specified file mode: `'r'`, `'w'`, `'a'`, or `'x'`. Python treats files as containing either text `'t'` or binary `'b'` data. Text mode is assumed. You must explicitly set the binary file mode. Add the updating mode `'+'` to open a file for both reading and writing.

The `open()` function returns a file object via which you interact with the file on disk. Use a context manager to automatically manage the file resource.

A relative file path is formed starting from a directory other than the root volume. The Unix/Linux file path separator is the forward slash `'/'`. The Windows file path separator is the backslash `'\'`. A Unix/Linux absolute file path is formed starting from the root volume which is indicated by a starting forward slash `'/'`. A Windows absolute file path begins with a drive letter.

Do not hard code file paths in Python programs. Use the `os` package to create and manipulate OS-agnostic file paths.

2 Binary Data and Random File I/O

Binary data and random file I/O go together like Saint Tropez et Bain de Soleil. In this section you will learn how binary data differs from string data, how to write binary data to a file, how to read binary data from a file, and how to conduct random file I/O operations with a binary data file. Let's start with binary data.

2.1 Binary Data

Binary data refers to the raw bytes that underlie all digital data. (*Refer to Chapter 4: Computers, Programs, and Algorithms, for a deeper discussion of bits, bytes, and words.*) The most important thing you need to remember is that all data on a computer is binary data.

2.1.1 Text Is Encoded Binary Data

Text is binary data that has been encoded to represent a particular character set, either ASCII or Unicode. Python text is represented as Unicode strings, which must be converted into a sequence of bytes via UTF-8 encoding before being written to disk or transmitted over a network. (*UTF-8 stands for Unicode Transformation Format — 8-Bit*) When writing text to a file this encoding is done automatically for you, but when you work directly with binary data, you, the programmer, need to decide what a sequence of bytes represents and treat them accordingly.

2.2 Writing and Reading Binary Data

Let's look as some code that works with binary data. Example 16.5 offers a short program that creates, writes, and reads a binary data file.

16.5 binary_files.py

```python
"""Demonstrate binary file I/O."""

import os

def main():
    # Create some binary data
    byte_array = bytearray([0b00000000, 0b00000001, 0b00000010, 0b00000011])
    hex_bytes = b'\x0f\x05\x15\xFF'

    # Setup path variables
```

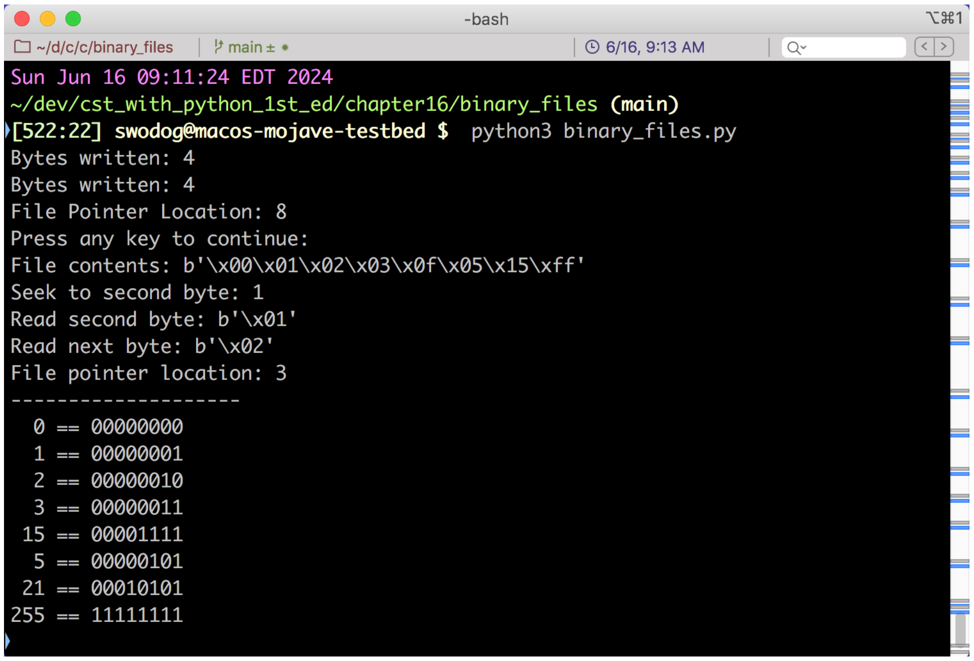
```
11      working_dir = os.getcwd()
12      data_dir = 'data'
13      data_dir_path = os.path.join(working_dir, data_dir)
14      data_file = 'binary.dat'
15
16      try:
17         if not os.path.exists(data_dir_path):
18            os.makedirs(data_dir_path)
19
20         with open(os.path.join(data_dir_path, data_file), 'wb') as f:
21            bytes_written = f.write(byte_array)
22            print(f'Bytes written: {bytes_written}')
23            bytes_written = f.write(hex_bytes)
24            print(f'Bytes written: {bytes_written}')
25            print(f'File Pointer Location: {f.tell()}')
26
27         input('Press any key to continue: ')
28
29         with open(os.path.join(data_dir_path, data_file), 'rb') as f:
30            # Opening a file for reading sets file pointer
31            # to beginning of file...
32            print(f'File contents: {f.read()}')
33            print(f'Seek to second byte: {f.seek(1)}')
34            print(f'Read second byte: {f.read(1)}')
35            print(f'Read next byte: {f.read(1)}')
36            print(f'File pointer location: {f.tell()}')
37
38            print('-' * 20)
39            # Reset file pointer to beginning of file
40            f.seek(0)
41            # Read file contents into byte array
42            file_bytes = bytearray(f.read())
43            # Iterate over each byte and print to console
44            for b in file_bytes:
45               print(f'{b:3} == {b:08b}')
46
47      except (OSError, Exception) as e:
48         print(f'Problem reading file: {e}')
49
50
51   if __name__ == '__main__':
52      main()
53
```

Referring to example 16.5 — On line 7, I create a variable named `byte_array` using the `bytearray()` constructor with four binary literals. Note that a binary literal has a prefix of `'0b'`. On line 8, I create another variable named `hex_bytes` initialized with a byte string literal consisting of hexadecimal values. Within the binary string, each hexadecimal value is prefixed with `'\x'`. On lines 11 through 14, I create a set of variables that configure the path to the current working directory, the name of the data directory, and the name of the data file. Then, in the `if` statement starting on line 17, I verify the existence of the `data` directory and create it if it's not there.

On lines 20 through 25, I open the file in `'wb'` (write binary) mode and write both the `byte_array` and `hex_bytes` data to the file and then print various statistics regarding the file operations, including how many bytes were written and the location of the file pointer after the last `f.write()` operation.

On line 27, I prompt the user to press any key to continue and then open the file in `'rb'` (read bytes) mode and on line 32 print the entire contents of the file to the console. After the first call to `f.read()`, the file pointer is at the end of the file. A subsequent call to `f.read()` at that location would return nothing, so, on line 33, I make a call to `f.seek(1)` to position the file pointer at the second byte within the file and then call `f.read(1)` to read one byte. This advances the file pointer by one byte at which point I read one more byte with `f.read(1)`. Then, on line 36, I call the `f.tell()` method to report the current position of the file pointer.

On line 38, I print the hyphen character `'-'` 20 times to create a section divider then reset the file pointer to the beginning of the file with a call to `f.seek(0)`, read the entire file with `f.read()`, and pass the results into the `bytearray()` constructor. I then iterate over each byte and print its decimal value followed by its binary representation. Figure 16-1 shows the results of running this program.



```
Sun Jun 16 09:11:24 EDT 2024
~/dev/cst_with_python_1st_ed/chapter16/binary_files (main)
[522:22] swodog@macos-mojave-testbed $  python3 binary_files.py
Bytes written: 4
Bytes written: 4
File Pointer Location: 8
Press any key to continue:
File contents: b'\x00\x01\x02\x03\x0f\x05\x15\xff'
Seek to second byte: 1
Read second byte: b'\x01'
Read next byte: b'\x02'
File pointer location: 3
--------------------
  0 == 00000000
  1 == 00000001
  2 == 00000010
  3 == 00000011
 15 == 00001111
  5 == 00000101
 21 == 00010101
255 == 11111111
```

Figure 16-9: Results of Running Example 16.5

Referring to figure 16-9 — A call to `f.seek()` returns the file pointer's new position with in the file. (i.e., `f.seek(1)` returns 1) Also, just like an array, a file's first addressable byte starts at location zero.

2.3 More About Seeking

Table 16-4 gives several examples of `seek()` method usage.

Seek Call	Meaning
`f.seek(0)`	Seek to the beginning of file. (i.e., Seek zero bytes from the beginning of the file.) Second argument defaults to zero.

Table 16-4: Example of Calling File `Seek()` Method

Seek Call	Meaning
`f.seek(-2, 1)`	Seek two bytes back from current file pointer location. Will raise an exception if there is not enough room to move backwards two bytes.
`f.seek(8)`	Seek eight bytes from beginning of file. Will raise an exception if there is not enough room in the file.
`f.seek(-10, 2)`	Seek backwards ten bytes from end of file. Will raise an exception if there is not enough space in the file.
`f.seek(0, 2)`	Seek to end of file.

Table 16-4: Example of Calling File `Seek()` Method

Referring to table 16-4 — A call to `f.seek()` moves the file pointer within the file relative from one of three positions: the *beginning* of the file, the *current position*, or the *end of the file*. These relative positions are indicated by the second argument to the `seek()` method which includes `0` (from beginning of file), `1` (from current position), or `2` (from end of file). Note that anytime you attempt to move the file pointer within a file it needs to have enough room to support the move. For example, seeking backwards from the beginning of the file will raise an exception, as will attempting to seek past the end of the file.

2.4 Binary Strings vs. Text

When you write binary to a file, the `write()` method returns the number of *bytes written*, whereas when writing text, the `write()` method returns the number of *characters written*. This may catch you off guard if you are expecting a 1-to-1 correspondence between writing a binary string and a regular string. In most cases, they will be the same, but if the regular string contains extended Unicode characters that cannot be represented by the Basic Latin character set, then the number of bytes written will be more than the number of characters written.

As you recall from the previous section, Python strings are Unicode. Most English speakers who have there computers configured to use English use the Basic Latin Character code set, which includes the ASCII characters in the first eight bits. When writing Basic Latin Unicode characters to disk or transmitting over the network, the characters are usually encoded into UTF-8, which drops the eight most-significant bits. These bits are all zeros anyway, so there's no loss of data, and it takes up less space on the disk.

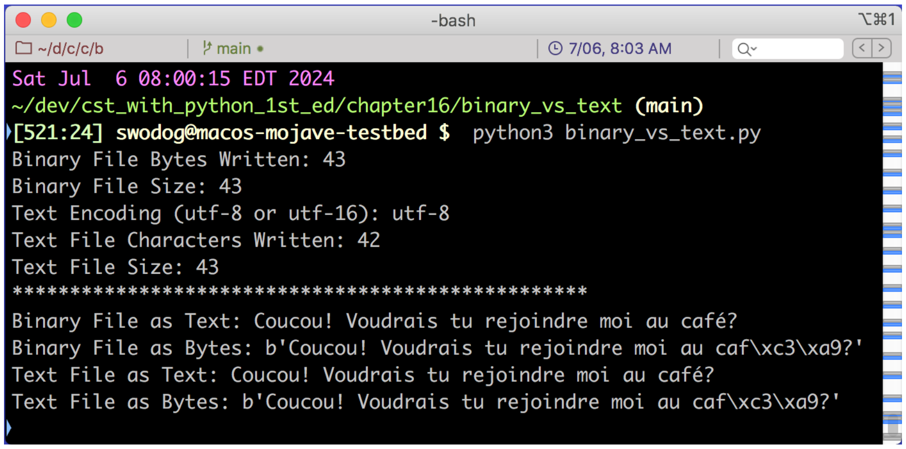
If you write text in Latin languages that include character accent marks like the French `'é'`, then both Unicode bytes will be written for that special character. Example 16.6 provides a short program that highlights the differences between writing text as text vs. text as binary.

16.6 binary_vs_text.py

```python
"""Demonstrate the difference between byte strings and text."""

import os

def main():
    byte_string = b'Coucou! Voudrais tu rejoindre moi au caf\xc3\xa9?'
    text_string = 'Coucou! Voudrais tu rejoindre moi au café?'

    working_dir = os.getcwd()
    data_dir = 'data'
    data_dir_path = os.path.join(working_dir, data_dir)
```

```
12     binary_filename = 'binary_file.dat'
13     text_filename = 'text_file.txt'
14
15     try:
16        if not os.path.exists(data_dir_path):
17           os.makedirs(data_dir_path)
18
19        # Write binary data
20        # Binary mode takes no encoding argument
21        binary_file_path = os.path.join(data_dir_path, binary_filename)
22        with open(binary_file_path, 'wb') as f:
23           print(f'Binary File Bytes Written: {f.write(byte_string)}')
24
25        # Print file size to console
26        print(f'Binary File Size: {os.path.getsize(binary_file_path)}')
27
28        # Get text encoding to use with text mode
29        encoding = input('Text Encoding (utf-8 or utf-16): ')
30        if encoding not in ['utf-8', 'utf-16']:
31           encoding = 'utf-8'
32
33        # Write text data
34        text_file_path = os.path.join(data_dir_path, text_filename)
35        with open(text_file_path, 'w', encoding=encoding) as f:
36           print(f'Text File Characters Written: {f.write(text_string)}')
37
38        # Print text file size to console
39        print(f'Text File Size: {os.path.getsize(text_file_path)}')
40
41        print('*' * 50)
42
43        # Read binary file as text
44        with open(binary_file_path, 'r') as f:
45           print(f'Binary File as Text: {f.read()}')
46
47        # Read binary file as bytes
48        with open(binary_file_path, 'rb') as f:
49           print(f'Binary File as Bytes: {f.read()}')
50
51        # Read text file as text
52        with open(text_file_path, 'r', encoding=encoding) as f:
53           print(f'Text File as Text: {f.read()}')
54
55        # Read text file as bytes
56        with open(text_file_path, 'rb') as f:
57           print(f'Text File as Bytes: {f.read()}')
58
59
60     except (OSError, Exception) as e:
61        print(f'Problem writing or reading files: {e}')
62
63
64  if __name__ == '__main__':
65     main()
66
```

Referring to example 16.6 — On lines 6 and 7, I define a binary string and text string. Each string conveys the same message in French which translates to "Hey! Would you like to join me at the café?" (*It's the accented* `'é'` *I am after for this example!*) Note that on line 6 that a byte string can only contain byte characters, which include ASCII, so the `'é'` must be expressed as a series of hex values. Next, the byte string is written to the binary file in binary mode. The user is then prompted to enter an encoding to use for writing the text file and can choose between UTF-8 or UTF-16. The text string is then written to the text file using the specified encoding. Finally, both files are opened and read in both binary and text modes and the results printed to the console as shown in figure 16-10.



```
Sat Jul  6 08:00:15 EDT 2024
~/dev/cst_with_python_1st_ed/chapter16/binary_vs_text (main)
[521:24] swodog@macos-mojave-testbed $  python3 binary_vs_text.py
Binary File Bytes Written: 43
Binary File Size: 43
Text Encoding (utf-8 or utf-16): utf-8
Text File Characters Written: 42
Text File Size: 43
**************************************************
Binary File as Text: Coucou! Voudrais tu rejoindre moi au café?
Binary File as Bytes: b'Coucou! Voudrais tu rejoindre moi au caf\xc3\xa9?'
Text File as Text: Coucou! Voudrais tu rejoindre moi au café?
Text File as Bytes: b'Coucou! Voudrais tu rejoindre moi au caf\xc3\xa9?'
```

Figure 16-10: Results of Running Example 16.6

Referring to figure 16-10 — Note that the number of characters written to the text file are less than the size of the text file. This is because the Unicode `'é'` requires two bytes as opposed to the ASCII characters, which only require one byte.

2.5 Reading Image Metadata

You can really have a lot of fun poking around files in binary mode. In this section, I want to give you a breathless introduction on how to read metadata from JPEG files. Now, there are 3rd party libraries that can read the metadata for you, but if you really want to know more about image file structure, nothing beats studying the specification and poking around the image file in binary mode. That was my motivation for working on this example — I wanted to know more about image files. But more importantly, the example in this section demonstrates how to conduct random binary file I/O to solve a real-world problem.

2.5.1 Background

JPEG images contain a mixture of image data and metadata. The metadata is referred to as EXIF (Exchangeable Image File Format) and is structured according to the Camera and Imaging Products Association standard CIPA-DC-008-2012 [*https://www.cipa.jp/std/documents/e/DC-008-2012_E.pdf*] After studying the specification and working on the code for this example, it became quite clear why my camera EXIF data gets clobbered by Adobe Photoshop.

2.5.2 Basic JPEG File Structure

In a nutshell, a JPEG file begins with a two-byte code that signals the Start-of-Image (SOI) `'FFD8'` and ends with a two-byte code that signals the End-of-Image (EOI) `'FFD9'`. The JPEG image may or may not contain EXIF data. If it does, it will be located in application segment

'APP0' identified by the two-byte code 'FFE1'. The data is written in a particular byte order indicated by either the two-byte code 'II' (INTEL) or 'MM' (MOTOROLA). If the byte order is INTEL then data is stored in Little Endian order meaning the least significant two bytes come before the most significant two-bytes. These bytes must be swapped to produce the correct value.

Once the byte-order is established, there's an entry that indicates how many EXIF tags are present. Each EXIF tag record consists of twelve bytes structured as shown in table 16-5.

Tag Record Field	Bytes
Tag	2
Tag Type	2
Data Length	4
Data/Offset	4

Table 16-5: EXIF Tag Record Structure

Referring to table 16-5 — Tags convey various types of data about the image, as you will see here shortly.

2.5.3 Example Code

OK, the example for this section consists of two source files, constants.py and main.py. Example 16.7 gives the listing for constants.py.

16.7 constants.py

```
"""JPEG and EXIF Tag and Marker definitions.

NOTE: This is not a complete list by any means.
"""

import types

# JPEG Markers and Segments
jpeg_markers = types.SimpleNamespace()
jpeg_markers.SOI = b'\xff\xd8'
jpeg_markers.EOI = b'\xff\xd9'
jpeg_markers.APP1 = b'\xff\xe1'
jpeg_markers.COMMENT = b'\xff\xfe'
jpeg_markers.EXIF = b'Exif'
jpeg_markers.INTEL = b'II*\x00'
jpeg_markers.MOTOROLA = b'MM\x00*'

# Tag Codes -- String version of 2-byte hex values
# Very small subset of EXIF tags
# Object properties can be used in match cases
tag_codes = types.SimpleNamespace()
tag_codes.image_width = '0100'
tag_codes.image_length = '0101'
tag_codes.bits_per_sample = '0102'
tag_codes.photometric_interpretation = '0106'
tag_codes.make = '010f'
tag_codes.model = '0110'
tag_codes.orientation = '0112'
tag_codes.samples_per_pixel = '0115'
tag_codes.x_resolution = '011a'
tag_codes.y_resolution = '011b'
tag_codes.resolution_unit = '0128'
tag_codes.software = '0131'
tag_codes.datetime = '0132'
tag_codes.artist = '013b'
tag_codes.host_computer = '013c'
tag_codes.ycbcrpositioning = '0213'
tag_codes.copyright = '8298'
tag_codes.exif_offset = '8769'
tag_codes.gps_info = '8825'
tag_codes.unknown = '0000'

# Tag Values
tags = {}
tags[tag_codes.image_width] = 'Image Width'
tags[tag_codes.image_length] = 'Image Length'
tags[tag_codes.bits_per_sample] = 'Bits Per Sample'
tags[tag_codes.photometric_interpretation] = 'Photometric Interpretation'
tags[tag_codes.make] = 'Make'
tags[tag_codes.model] = 'Model'
tags[tag_codes.orientation] = 'Orientation'
tags[tag_codes.samples_per_pixel] = 'Samples Per Pixel'
tags[tag_codes.x_resolution] = 'X-Resolution'
tags[tag_codes.y_resolution] = 'Y-Resolution'
tags[tag_codes.resolution_unit] = 'Resolution Unit'
tags[tag_codes.software] = 'Software'
tags[tag_codes.datetime] = 'DateTime'
tags[tag_codes.artist] = 'Artist'
tags[tag_codes.host_computer] = 'Host Computer'
tags[tag_codes.ycbcrpositioning] = 'YCbCrPositioning'
tags[tag_codes.copyright] = 'Copyright'
tags[tag_codes.exif_offset] = 'EXIF Offset'
tags[tag_codes.gps_info] = 'GPS Info'
tags[tag_codes.unknown] = 'Unknown'

# Tag Types
tag_types = {}
tag_types[1] = ('Unsigned Byte')
tag_types[2] = ('ASCII String')
tag_types[3] = ('Unsigned Short')
tag_types[4] = ('Unsigned Long')
tag_types[5] = ('Unsigned Rational')
tag_types[6] = ('Signed Byte')
tag_types[7] = ('Undefined')
tag_types[8] = ('Signed Short')
tag_types[9] = ('Signed Long')
tag_types[10] = ('Signed Rational')
tag_types[11] = ('Single')
tag_types[12] = ('Double')
tag_types[129] = ('UTF-8')

# Orientation Values
orientation = {}
orientation[1] = 'Horizontal (Normal)'
orientation[2] = 'Horizontal Mirrored'
orientation[3] = 'Rotated 180 Degrees'
orientation[4] = 'Vertical Mirrored'
```

```python
orientation[5] = 'Horizontal Mirrored then Rotated 190 Degrees CCW'
orientation[6] = 'Rotated 90 Degrees CW'
orientation[7] = 'Horizontal Mirrored then Rotated 90 Degrees CW'
orientation[8] = 'Rotated 90 Degrees CCW'

# Resolution Unit Values
resolution_unit = {}
resolution_unit[1] = 'Not Absolute'
resolution_unit[2] = 'Pixels/Inch'
resolution_unit[3] = 'Pixels/Centimeter'

```

Referring to example 16.7 — As the doc comment indicates, this is by no means a complete listing. It represents the tags and tag types I encountered while developing this example. Note that I am importing the `types` package and using the `types.SimpleNamespace()` on lines 9 and 21 to create namespaces to hold the `jpeg_markers` and `tag_codes` objects. I needed to do this so I could use these values in match/case statements. Example 16-8 gives the listing for main.py.

16.8 main.py

```python
"""Demonstrate reading binary image EXIF data."""

import os
from constants import jpeg_markers
from constants import tag_codes
from constants import tag_types
from constants import tags
from constants import orientation
from constants import resolution_unit


def main():
    try:
        # Setup file paths
        working_dir = os.getcwd()
        image_dir = 'images'
        image_dir_path = os.path.join(working_dir, image_dir)

        # Create image directory if it does not exist
        if not os.path.exists(image_dir_path):
            os.makedirs(image_dir_path)

        # Get JPEG image filename from user
        filename = input('JPEG Image Filename: ')

        # Open JPEG file in read binary mode
        with open(os.path.join(image_dir_path, filename), 'rb') as f:

            # Read entire file
            content = f.read()

            # Print first 512 bytes and print hex
            for s in content[:512]:
                print(f'{s:02x} ', end='')
            print()

            # Read  forst 512 bytes and print chars
            for s in content[:512]:
                print(f'{chr(s)} ', end='')
            print()
```

```

      # Reset file pointer to beginning of file
      f.seek(0)

      # Verify it's a JPEG file
      print(f'Verifying JPEG File...')
      if is_jpeg_file(content[:2], content[-2:]):
         print(f'{filename} is a JPEG file. Extracting EXIF data...')
      else:
         print(f'{filename} is not a JPEG file. Exiting...')
         return


      # Find Segment Offsets
      print(f'{"-" * 10} Segment Offsets {"-" * 10}')

      app1_segment_offset = None
      try:
         app1_segment_offset = content.index(jpeg_markers.APP1)
         print(f'APP1 Segment offset: {app1_segment_offset}')
      except Exception:
         print(f'APP1 segment not found.')

      exif_segment_offset = None
      try:
         exif_segment_offset = content.index(jpeg_markers.EXIF)
         print(f'Exif Segment offset: {exif_segment_offset}')
      except Exception:
         print('Exif segment not found.')

      comment_segment_offset = None
      try:
         comment_segment_offset = content.index(jpeg_markers.COMMENT)
         print(f'Comment Segment offset: {comment_segment_offset}')
      except Exception:
         print('Comment segment not found.')

      # Exit if no EXIT segment
      if not exif_segment_offset:
         print(f'{filename} does not contain an EXIF segment. ')
         print('Exiting program.')
         return

      # Determine Endian
      endian_offset = None
      intel = False
      try:
         endian_offset = f.seek(exif_segment_offset + 6)
         endian_marker = f.read(4)
         if endian_marker == jpeg_markers.INTEL:
            intel = True
            print(f'Endian Marker: {endian_marker} : Endian is INTEL')
         else:
            print(f'Endian Marker: {endian_marker} : Endian is MOTOROLA')

      except Exception:
         print('Endian offset not found.')

      # Print the endian marker
```

```python
            f.seek(endian_offset)
            print(f'Endian Offset: {endian_offset} Read 4 : {f.read(4)}')
            f.seek(endian_offset + 8)
            exif_entries_raw_bytes = bytearray(f.read(2))

            # How many EXIF records are there?
            print(f'Exif Entries Raw Bytes: {bytes(exif_entries_raw_bytes)}')
            f.seek(-2, 1)
            exif_entries = int(swap_bytes(f.read(2)))
            print(f'Exif Entries: {exif_entries}')

            print('*' * 130)
            # Print column headers
            print(f'{"No.":<8}{"Raw Bytes":<10}{"Tag Hex":<10}\
{"Tag Name":30}{"Tag Type":<20} {"Length":<8}{"Data/Offset":<15}{"Data":<50}')
            print('-' * 130)

            # Read Tag Records
            # Every 12 bytes from just past exif_entries bytes.
            #
            # ----Tag Record Layout----
            # Tag:         2 Bytes
            # Tag Type:    2 Bytes
            # Data Length: 4 Bytes
            # Data/Offset: 4 Bytes
            ###########################

            for i in range(exif_entries):
                # Read tag bytes
                tag = bytearray(f.read(2))

                # Convert to hex string
                tag_hex = 0
                if intel:
                    tag_hex = f'{tag[1]:02x}{tag[0]:02x}'

                else:
                    tag_hex = f'{tag[0]:02x}{tag[1]:02x}'

                # Read Tag Type
                tag_type = int(swap_bytes(f.read(2)))
                # Read Data Length
                data_length = \
                    int(swap_bytes(f.read(2)) + swap_bytes(f.read(2)))
                # Read data or offset
                data_or_offset = \
                    int(swap_bytes(f.read(2)) + swap_bytes(f.read(2)))

                data = 'None'
                if data_length > 4:
                    last_position = f.tell()
                    f.seek(endian_offset + data_or_offset)
                    data = f.read(data_length)
                    f.seek(last_position)
                elif data_length == 1:
                    match tag_hex:
                        case tag_codes.orientation:
                            data = orientation[data_or_offset]
                        case tag_codes.resolution_unit:
```

```
159                       data = resolution_unit[data_or_offset]
160                    case _: pass
161
162             # Print EXIF data
163             print(f'{i: <8d}{f"{tag[0]:02x}{tag[1]:02x}":10}\
164 {tag_hex: <10}{tags.get(tag_hex):30}{tag_types.get(tag_type):<20}\
165 {data_length:<8d}{data_or_offset:<15}{data}')
166
167
168
169    except (OSError, Exception) as e:
170       print(f'Problem reading image file: {e}')
171
172
173 # Utility Methods
174 def is_jpeg_file(first_two_file_bytes:bytes,
175               last_two_file_bytes:bytes)->bool:
176    """Verify JPEG SOI and EOI."""
177    if __debug__:
178       print(f'SOI: {first_two_file_bytes} : \
179       EOI: {last_two_file_bytes}')
180    return (first_two_file_bytes == jpeg_markers.SOI) \
181       and (last_two_file_bytes == jpeg_markers.EOI)
182
183
184 def swap_bytes(b:bytes)-> bytes:
185    """Swap little endian bytes."""
186    return b[1] + b[0]
187
188
189 if __name__ == '__main__':
190    main()
191
```

Referring to example 16.8 — It's probably a lot easier to load this file in Visual Studio Code and follow along that way rather than flipping pages back and forth. The first thing this program does is load a file, read its entire contents, and prints the first 512 bytes to the console in two formats: hexadecimal and ASCII. Next, I verify the file is a JPEG file by checking the SOI and EOI bytes. If it's not a JPEG file, the program exits. Next, on lines 57 through 76, I look for various JPEG segment markers and if they are present, I get their offsets. Note that this is the offset from the beginning of the file. If the EXIF segment is not present the program exits.

The next several lines of code determine the byte order marker (BOM) or *endianness* of the file, i.e, `II` (Intel or little endian) or `MM` (Motorola or big endian). From this point forward, the BOM determines how the EXIF data is processed. In the program output I show the raw byte order so you can see the difference between big endian vs. little endian byte order.

Next, eight bytes past the offset to the BOM resides two bytes that indicate the number of EXIF records. From this point forward, every twelve bytes contains an EXIF record in the format given in table 16-5. The `for` loop beginning on line 127 iterates over each EXIF record by reading each set of field bytes and processing them accordingly. Note that for the sake of the example, I am performing some extra byte processing so you can see how they are accessed and transformed.

Tag processing requires first reading the first two EXIF record bytes and swapping them if the file BOM is II (Intel) to get the tag value. I then look up the tag name and the tag type. Note that with the exception of the tag bytes, all data bytes are in little endian and must be swapped to obtain the correct values.

OK, now we need an image to check. I’ll use Cat.jpg shown in figure 16-11.

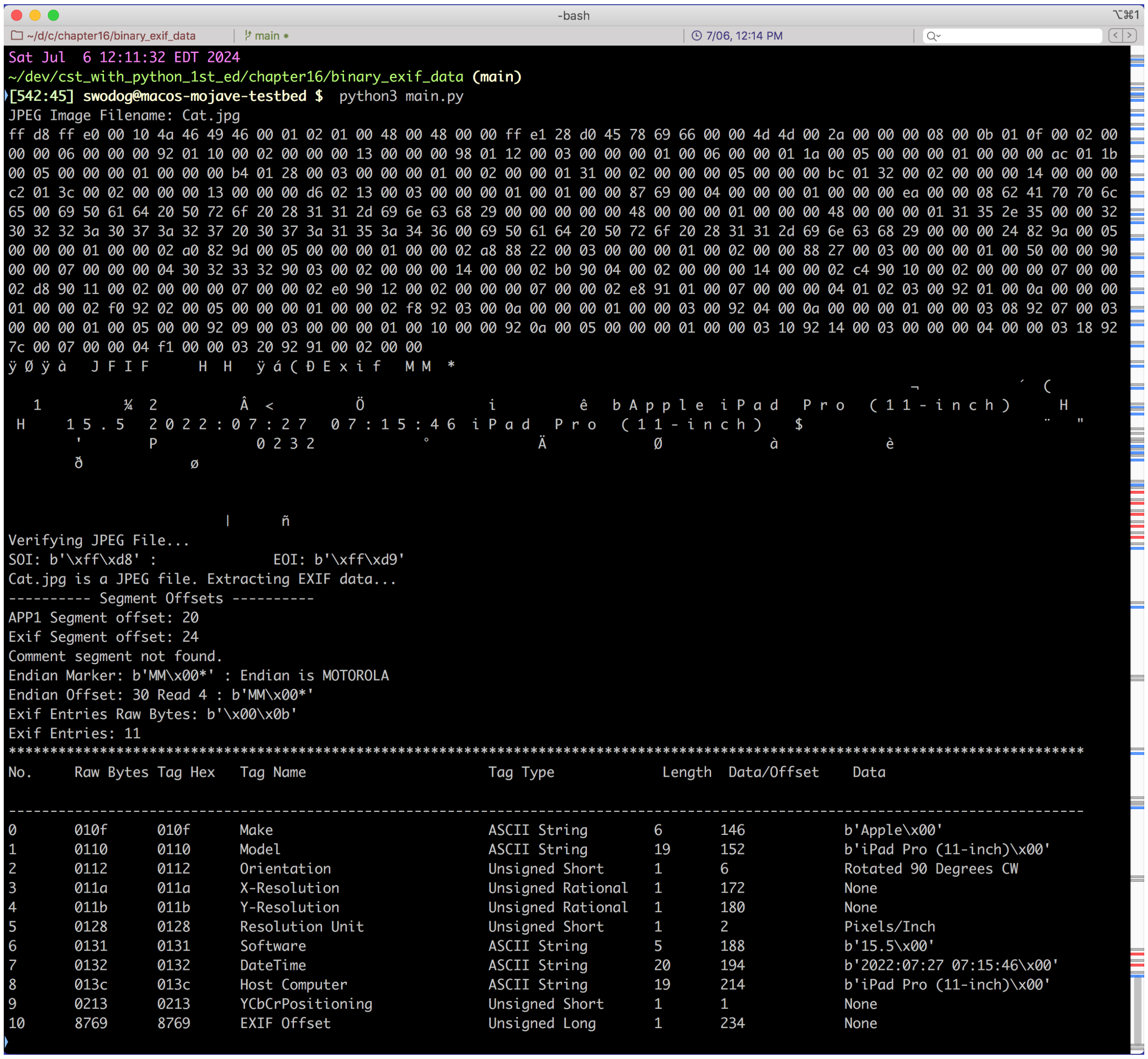
Figure 16-11: A Good Candidate Image — Cat.jpg

Referring to figure 16-11 — I took this image of our good friend Alice Findler’s cat on my iPad. Running the program on this image gives the results shown in figure 16-12.

Referring to figure 16-12 — The first 512 bytes are printed in hexadecimal then in ASCII. The ASCII version allows you to easily pick out the metadata. Note the letters JFIF which stand for JPEG File Interchange Format. In the hex output you can see the first two bytes are `'ffd8'` which is the JPEG SOI marker.

The next section of output shows the offset values to various JPEG segments, if they are present, and the endian marker, which in this case is MM. (Note the full marker is `'MM\x00*'`) The number of EXIF entries is determined, which in this case is 11, and then each marker is processed and the results printed in table form to the console.

Note that I am not currently processing all the EXIF data, so some of the entries in the Data column will show `None`. Look at the Length column entries. Any length over 4 means that the



```
Sat Jul  6 12:11:32 EDT 2024
~/dev/cst_with_python_1st_ed/chapter16/binary_exif_data (main)
[542:45] swodog@macos-mojave-testbed $  python3 main.py
JPEG Image Filename: Cat.jpg
ff d8 ff e0 00 10 4a 46 49 46 00 01 02 01 00 48 00 48 00 00 ff e1 28 d0 45 78 69 66 00 00 4d 4d 00 2a 00 00 00 08 00 0b 01 0f 00 02 00
00 00 06 00 00 00 92 01 10 00 02 00 00 00 13 00 00 00 98 01 12 00 03 00 00 00 01 00 06 00 00 01 1a 00 05 00 00 00 01 00 00 00 ac 01 1b
00 05 00 00 00 01 00 00 00 b4 01 28 00 03 00 00 00 01 00 02 00 00 01 31 00 02 00 00 00 05 00 00 00 bc 01 32 00 02 00 00 00 14 00 00 00
c2 01 3c 00 02 00 00 00 13 00 00 00 d6 02 13 00 03 00 00 00 01 00 01 00 00 87 69 00 04 00 00 00 01 00 00 00 ea 00 00 08 62 41 70 70 6c
65 00 69 50 61 64 20 50 72 6f 20 28 31 31 2d 69 6e 63 68 29 00 00 00 00 00 48 00 00 00 01 00 00 00 48 00 00 00 01 31 35 2e 35 00 00 32
30 32 32 3a 30 37 3a 32 37 20 30 37 3a 31 35 3a 34 36 00 69 50 61 64 20 50 72 6f 20 28 31 31 2d 69 6e 63 68 29 00 00 00 24 82 9a 00 05
00 00 00 01 00 00 02 a0 82 9d 00 05 00 00 00 01 00 00 02 a8 88 22 00 03 00 00 00 01 00 02 00 00 88 27 00 03 00 00 00 01 00 50 00 00 90
00 00 07 00 00 00 04 30 32 33 32 90 03 00 02 00 00 00 14 00 00 02 b0 90 04 00 02 00 00 00 14 00 00 02 c4 90 10 00 02 00 00 00 07 00 00
02 d8 90 11 00 02 00 00 00 07 00 00 02 e0 90 12 00 02 00 00 00 07 00 00 02 e8 91 01 00 07 00 00 00 04 01 02 03 00 92 01 00 0a 00 00 00
01 00 00 02 f0 92 02 00 05 00 00 00 01 00 00 02 f8 92 03 00 0a 00 00 00 01 00 00 03 00 92 04 00 0a 00 00 00 01 00 00 03 08 92 07 00 03
00 00 00 01 00 05 00 00 92 09 00 03 00 00 00 01 00 10 00 00 92 0a 00 05 00 00 00 01 00 00 03 10 92 14 00 03 00 00 00 04 00 00 03 18 92
7c 00 07 00 00 04 f1 00 00 03 20 92 91 00 02 00 00
ÿ Ø ÿ à   J F I F     H  H   ÿ á ( Ð E x i f   M M  *
                                                                                                     ¬              ´  (
   1         ¼  2          Â  <          Ö            i         ê  b A p p l e  i P a d   P r o   ( 1 1 - i n c h )     H
 H    1 5 . 5   2 0 2 2 : 0 7 : 2 7   0 7 : 1 5 : 4 6  i P a d   P r o   ( 1 1 - i n c h )   $                    ¨   "
       '         P           0 2 3 2          °             Ä             Ø             à             è
       ð             ø



                          |      ñ
Verifying JPEG File...
SOI: b'\xff\xd8' :                EOI: b'\xff\xd9'
Cat.jpg is a JPEG file. Extracting EXIF data...
---------- Segment Offsets ----------
APP1 Segment offset: 20
Exif Segment offset: 24
Comment segment not found.
Endian Marker: b'MM\x00*' : Endian is MOTOROLA
Endian Offset: 30 Read 4 : b'MM\x00*'
Exif Entries Raw Bytes: b'\x00\x0b'
Exif Entries: 11
*************************************************************************************************************************************
No.     Raw Bytes Tag Hex   Tag Name                      Tag Type            Length  Data/Offset    Data

-------------------------------------------------------------------------------------------------------------------------------------
0       010f      010f      Make                          ASCII String        6       146            b'Apple\x00'
1       0110      0110      Model                         ASCII String        19      152            b'iPad Pro (11-inch)\x00'
2       0112      0112      Orientation                   Unsigned Short      1       6              Rotated 90 Degrees CW
3       011a      011a      X-Resolution                  Unsigned Rational   1       172            None
4       011b      011b      Y-Resolution                  Unsigned Rational   1       180            None
5       0128      0128      Resolution Unit               Unsigned Short      1       2              Pixels/Inch
6       0131      0131      Software                      ASCII String        5       188            b'15.5\x00'
7       0132      0132      DateTime                      ASCII String        20      194            b'2022:07:27 07:15:46\x00'
8       013c      013c      Host Computer                 ASCII String        19      214            b'iPad Pro (11-inch)\x00'
9       0213      0213      YCbCrPositioning              Unsigned Short      1       1              None
10      8769      8769      EXIF Offset                   Unsigned Long       1       234            None
```

Figure 16-12: Results of Running Example 16.8 on Cat.jpg

Data/Offset column entries are offsets. These are mostly ASCII type tags as you can see from the table. Referring again to example 16.8 — the `if` statement starting on line 149 checks the value of the `data_length` field and if it's greater that 4 it must perform a jump to the offset location indicated by the `data_or_offset` field and read the number of characters indicated by the `data_length` field. Note that before the jump, the `last_position` is preserved with a call to `f.tell()` and restored after the jump with a call to `f.seek(last_position)`.

2.5.4 Parting Thoughts

I will remind you once again, this example is incomplete, but does provide valuable insight into how complex binary data is processed. Note that image processing is challenging because different vendors often write metadata to image files in proprietary formats.

Quick Review

Binary data underlies all data on a computer system. To write binary data to a file, open the file in write binary `'wb'` mode. To read binary data from a file, open the file in read binary `'rb'` mode. You cannot read or write text (ordinary strings) to a file opened for reading or writing binary data and vice versa. A file opened in a binary mode cannot take an encoding argument.

Binary files support random file I/O. Use the `seek()` method to move the file pointer to a desired location. Use the `read()` method to read bytes, and use the `tell()` method to obtain the file pointer's current position.

3 Serializing Objects To File With Pickle

Pickle is a Python library package that lets you serialize objects to a file. Serialization is the process of converting objects in memory to a format that can be saved to a file or transmitted over the network. An object serialized with `pickle` can be deserialized with `pickle`. Deserialization is the process of reconstituting a serialized object back into an object in memory.

In this section, I will show you how to serialize a dictionary with `pickle` and save it to a file. I will also show you how to use Python's `hmac` and `hashlib` packages to generate a message digest from the serialized object which can be used to verify the integrity of the of the deserialized data. Example 16.9 gives a short example of how to pickle a dictionary.

16.9 pickle_demo.py

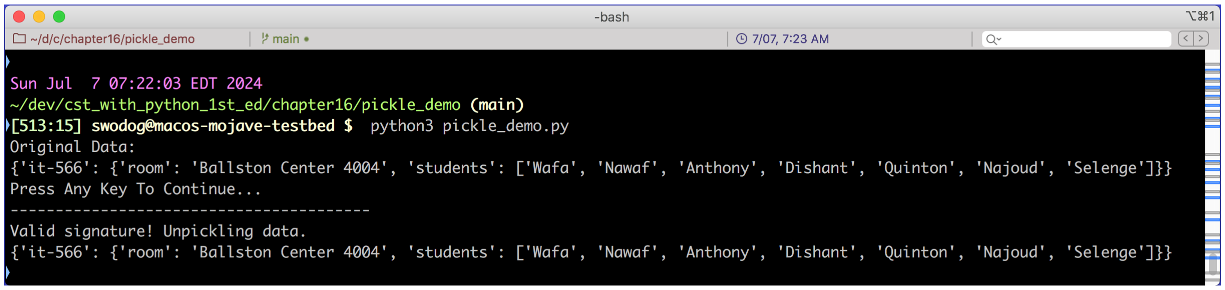
```
"""Demonstrate object serialization to file with Pickle package."""

import pickle
import hmac
import hashlib
import os

def main():
   try:
      # Setup path variables
      working_dir = os.getcwd()
      data_dir = 'data'
      data_dir_path = os.path.join(working_dir, data_dir)
      filename = 'classes.dat'

      if not os.path.exists(data_dir_path):
         os.mkdir(data_dir_path)

      # Some data pickle
      classes = {}
      classes['it-566'] = {}
      classes['it-566']['room'] = 'Ballston Center 4004'
      classes['it-566']['students'] = ['Wafa', 'Nawaf',
                          'Anthony', 'Dishant', 'Quinton',
                          'Najoud', 'Selenge']
      # Print to console
      print(f'Original Data:\n{classes}')

      # Pickle
      pickled_data = pickle.dumps(classes)

```

```
32         # Calculate digest signature to verify no tampering
33         expected_digest = hmac.new(b'some_key_value', pickled_data, \
34                           hashlib.sha256).hexdigest()
35
36         # Write to file
37         with open(os.path.join(data_dir_path, filename), 'wb') as f:
38            f.write(pickled_data)
39
40         # Press any key to continue
41         input('Press Any Key To Continue...')
42         print('-' * 40)
43
44         # Read file and verify signature
45         pickled_data = None
46         classes = None
47         with open(os.path.join(data_dir_path, filename), 'rb') as f:
48            pickled_data = f.read()
49
50         digest = hmac.new(b'some_key_value', pickled_data, \
51                   hashlib.sha256).hexdigest()
52
53         if not hmac.compare_digest(digest, expected_digest):
54            print(f'Invalid signature! Data is invalid.')
55            exit()
56         else:
57            print(f'Valid signature! Unpickling data.')
58            classes = pickle.loads(pickled_data)
59            print(f'{classes}')
60
61      except (OSError, Exception) as e:
62         print(f'Problem pickling data: {e}')
63
64
65   if __name__ == '__main__':
66      main()
67
```

Referring to example 16.9 — First, I import the `pickle`, `hmac`, `hashlib`, and `os` packages. Next, in the body of the `main()` function, I set up the data file paths and create a dictionary named `classes` and populate it with classroom and student names. Then, on line 30, I call `pickle.dumps()` to serialize the classes dictionary.

To calculate the message digest, I call the `hmac.hexdigest()` method. Note the arguments passed to the `hmac.new()` method. The first argument is a byte string `b'some_key_value'`. In the wild, you would want to use a unique key and safeguard it to prevent unauthorized use. The second argument is the message or data to use to generate the digest, and finally, the third argument is the encryption algorithm. I have assigned the results to a variable named `expected_digest` to denote that this is the value I expect when I later read the bytes from the file to ensure it has not been tampered with. I then write the pickled bytes to the file.

To verify the file's integrity, I read the file and calculate the message digest as before, then compare the `digest` with the `expected_digest` using the `hmac.compare_digest()` method. If the two values are identical, I deserialize the object with the `pickle.loads()` method. Figure 16-13 shows the results of running this program.



```
Sun Jul  7 07:22:03 EDT 2024
~/dev/cst_with_python_1st_ed/chapter16/pickle_demo (main)
[513:15] swodog@macos-mojave-testbed $  python3 pickle_demo.py
Original Data:
{'it-566': {'room': 'Ballston Center 4004', 'students': ['Wafa', 'Nawaf', 'Anthony', 'Dishant', 'Quinton', 'Najoud', 'Selenge']}}
Press Any Key To Continue...
----------------------------------------
Valid signature! Unpickling data.
{'it-566': {'room': 'Ballston Center 4004', 'students': ['Wafa', 'Nawaf', 'Anthony', 'Dishant', 'Quinton', 'Najoud', 'Selenge']}}
```

Figure 16-13: Results of Running Example 16.9

3.1 You May Be In A Pickle

Pickle provides a quick, handy way to serialize objects for either saving to disk or transmitting over the network. The problem with pickle is that it's Python proprietary, meaning objects serialized with pickle can only be deserialized with Python and pickle. Well, not so fast. A quick Google search yielded the Pyrolite project on GitHub [*https://github.com/irmen/Pyrolite*] which provides pickle capabilities to the Java and C# languages. Still, if sharing data is your intent, don't use pickle to save your data. Use JSON, XML, or CSV files instead.

If you really want to share binary serialized data between programming languages, use Google's language neutral Protocol Buffers. [*https://protobuf.dev*]

Quick Review

The `pickle` package provides a quick, easy way to serialize Python objects to they can be saved to disk or transmitted over a network. Serialization is the process by which an object is converted from its memory representation into a format suitable for saving to file or network transmission. Deserialization is the process by which a serialized object is reconstituted into its memory representation. Pickle is Python centric and best used by Python programs. If you really need to share serialized binary data between different programming languages used Google's Protocol Buffers.

4 Saving JSON Data To File

Saving JSON data to file is a cinch. I have already introduced you to JSON in previous chapters. Once you have your JSON string you can simply write it to file in text mode as shown in example 16.10.

16.10

```python
"""Demonstrate saving and reading JSON data to file."""

import json
import os

def main():
    # Create dictionary with data
    classes = {}
    classes['it-590'] = {}
    classes['it-590']['room'] = 'Ballston Center 3066'
    classes['it-590']['students'] = ['Davis', 'Lewis', 'Quinton']

    print(f'Classes Dictionary:\n{classes}')

    try:
        # Set up file paths
        working_dir = os.getcwd()
        data_dir = 'data'
        data_dir_path = os.path.join(working_dir, data_dir)
        filename = 'classes.json'

        # Create data directory if it does not exist
        if not os.path.exists(data_dir_path):
            os.mkdir(data_dir_path)

        # Convert data to json
        json_string = json.dumps(classes)
        print(f'Classes JSON:\n{json_string}')

        # Write json to file
        with open(os.path.join(data_dir_path, filename), 'w') as f:
            f.write(json_string)

        # Press any key to continue
        input('Press any key to continue...')
        print('-' * 40)

        # Read the json file and convert back into dictionary
        classes = None
        json_string = None
        with open(os.path.join(data_dir_path, filename), 'r') as f:
            print(f'Reading JSON from file...')
            json_string = f.read()
            print(f'Converting JSON string to Dictionary...')
            classes = json.loads(json_string)

        # Print the dictionary and JSON string
        print(f'JSON String:\n{json_string}')
        print(f'Classes Dictionary:\n{classes}')

    except (OSError, Exception) as e:
        print(f'Problem with file I/O: {e}')


if __name__ == '__main__':
    main()

```

Referring to example 16.10 — From the top, I import the Python library `json` package, create a dictionary named classes and populate it with data, then convert the dictionary into a JSON string using the `json.dumps()` method. This creates a JSON string, which is then written to a text file. To reconstitute the dictionary, open the JSON file in text mode, read the entire file, then convert the JSON string back into a dictionary with the `json.loads()` method. I must admit, the hardest part about all this `dumps()` and `loads()` stuff is remembering which method does what! Notice these methods perform essentially the same actions as the `pickle.dumps()` and `pickle.loads()` methods, that is, `json.dumps()` serializes a Python object into a JSON text string, and `json.loads()` deserializes a JSON test string back into a Python object.

4.0.1 JSON Serialization and Deserialization

One important thing to keep in mind about serializing Python objects into JSON is that the `json.dumps()` method only works on fundamental Python types. If you try to convert a user-defined type into JSON you will need to implement custom serialization or add a `to_json()` method that provides the desired JSON representation of the object. I'll show you how to do this later in the book when you learn about classes and object-oriented programming.

4.0.2 Parting Thoughts

Note that since JSON is based on key/value pairs, it's natural to start by creating a Python dictionary and populate it to contain the data you want to convert to JSON. Use only fundamental Python types as values within the dictionary to ensure maximum compatibility when sharing data between systems.

Quick Review

To convert Python dictionaries to JSON import the `json` package and use the `json.dumps()` method. To convert the JSON string back into a dictionary use the `json.loads()` method.

JSON is text. To save a JSON string to a file open the file for writing in text mode `'w'`. To read a JSON file, open the file in read text mode `'r'`, read the entire file, and convert it into a dictionary.

Summary

Python makes it easy to work with files. You can do a lot with just a little bit of code. Use the built-in `open()` function to open a file in a specified file mode: `'r'`, `'w'`, `'a'`, or `'x'`. Python treats files as containing either text `'t'` or binary `'b'` data. Text mode is assumed. You must explicitly set the binary file mode. Add the updating mode `'+'` to open a file for both reading and writing.

The `open()` function returns a file object via which you interact with the file on disk. Use a context manager to automatically manage the file resource.

A relative file path is formed starting from a directory other than the root volume. The Unix/Linux file path separator is the forward slash `'/'`. The Windows file path separator is the backslash `'\'`. A Unix/Linux absolute file path is formed starting from the root volume which is indicated by a starting forward slash `'/'`. A Windows absolute file path begins with a drive letter.

Do not hard code file paths in Python programs. Use the `os` package to create and manipulate OS-agnostic file paths.

Binary data underlies all data on a computer system. To write binary data to a file, open the file in write binary `'wb'` mode. To read binary data from a file, open the file in read binary `'rb'` mode. You cannot read or write text (ordinary strings) to a file opened for reading or writing binary data and vice versa. A file opened in a binary mode cannot take an encoding argument.

Binary files support random file I/O. Use the `seek()` method to move the file pointer to a desired location. Use the `read()` method to read bytes, and use the `tell()` method to obtain the file pointer's current position.

The `pickle` package provides a quick, easy way to serialize Python objects to they can be saved to disk or transmitted over a network. Serialization is the process by which an object is converted from its memory representation into a format suitable for saving to file or network transmission. Deserialization is the process by which a serialized object is reconstituted into its memory representation. Pickle is Python centric and best used by Python programs. If you really need to share serialized binary data between different programming languages used Google's Protocol Buffers.

To convert Python dictionaries to JSON import the `json` package and use the `json.dumps()` method. To convert the JSON string back into a dictionary use the `json.loads()` method.

JSON is text. To save a JSON string to a file open the file for writing in text mode `'w'`. To read a JSON file, open the file in read text mode `'r'`, read the entire file, and convert it into a dictionary.

Skill-Building Exercises

1. **File Reading:** Write a Python script that reads the contents of a text file named `'example.txt'` and prints its contents to the console.

2. **File Writing:** Write a Python script that writes a list of strings to a file named `'output.txt'` with each string on a new line. Open the file with a text editor and inspect the contents.

3. **Appending To A File:** Modify the program you created in exercise two above and append strings to the file without overwriting the existing file.

4. **Read a File Line-by-Line:** Write a Python script that reads each line of `'output.txt'` created in the previous example one at a time and prints it to the console.

5. **Counting Lines, Words, and Characters:** Write a Python script that reads `'output.txt'` and prints the number of lines, words, and characters contained within the file.

6. **Copying File Content:** Write a Python script that copies the contents of `'output.txt'` to a file named `'copy.txt'`.

7. **Searching For A String:** Write a script that searches a text file for a particular string and returns the lines that contain the string.

8. **`JSON File Ops`:** Write a script that reads a JSON file, converts it into a dictionary, modifies the data in some way, then converts the dictionary into a JSON string and saves it to the same file.

9. **Binary File Ops:** Write a program that reads an image file you provide and prints the first N number of bytes in hexadecimal and ASCII format. The image file can be of any image type: JPEG, PNG, TIFF, etc.

10. **CSV File Processing:** Study the Python csv package. Write a program that reads a CSV file and prints the data contained within it to the console. Print a count of the number of items in each column and determine the type of data within each column. If the column data is numeric, print the sum and average of the data.

Suggested Projects

1. **Verify Image Files:** Write a program that scans a designated directory for image files and validates the type of each image file by reading the first two bytes of each file. **Note:** You will need to study each of the image file formats to learn what combination of bytes comprise the Start of Image (SOI) sequence.

2. **Simple Text File Editor:** Write a simple text file editor that lets you create, open, edit, and save text files.

3. **Image Metadata Extractor:** Research the Python `pillow` package. Write a program that lets you extract metadata from images using the `pillow` package.

4. **Binary Data Compression Tool:** Write a program that enables the user to compress and decompress binary files using Huffman coding or Run-Length Encoding (RLE).

5. **Working with Zip Files:** Research the Python `zipfile` package. Write a program that zips the contents of a designated directory.

6. **Working with Zip Files:** Write a program that lists the contents of a zip file and allows the user to choose one or files for extraction.

7. **Diary Entry Application:** Write a program that lets users create dated diary entries. Save each entry as a separate line in a text file. Allow users to filter and print diary entries by date.

8. **Expense Tracker:** Write a program that lets users enter, filter, and print expenses. The program should be able to save expense data to a file. Consider carefully how you will represent expenses within the program and the structure the data will take within the file. Ensure the expense data stored within the file is cross-platform compatible and can be easily shared with other applications.

9. **Expense Export Utility:** Write a program that reads a saved expenses file created in Suggested Project 8 above and exports it to CSV format for import into Microsoft Excel.

10. **Test File Merge Utility:** Write a program that lets the user select one or more text files and merge them into one text file. Order the merged content by file date.

Self-Test Questions

1. What two primary types of data can be written to a file?

2. What is the default file mode if none is provided to the `open()` function?

3. What method can you use to check for the existence of a file?

4. (True/False) You can write Python strings (str) to a file opened in write binary `'wb'` mode.

5. What does the `write()` method return when writing to a binary file?

6. What does the `write()` method return when writing to a text file?

7. What is a file pointer?

8. At what position does the file pointer start at when you open a file in `'w'` or `'wb'` modes?

9. What position does the file pointer indicate after reading the entire contents of a file?

10. What value does the `tell()` method return?

References

Python Documentation, Python Built-In open() Function: *https://docs.python.org/3/library/functions.html#open*

CIPA DC-008-Translation-2012 PDF: *https://www.cipa.jp/std/documents/e/DC-008-2012_E.pdf*

Python Documentation, File Objects: *https://docs.python.org/3/c-api/file.html*

Python Documentation, The Python Standard Library: *https://docs.python.org/3/library/index.html*

The Code Project, Understanding and Reading Exif Data: *https://www.codeproject.com/Articles/47486/Understanding-and-Reading-Exif-Data*

Python Documentation, Core Tools For Working with Streams: *https://docs.python.org/3/library/io.html*

Notes