Chapter 19
Stream IO

19.1 Introduction

In this chapter we will explore the Stream I/O model that under pins the way in which data is read from and written to data sources and sinks. One example of a data source or sink is a file but another might be a byte array.

This model is actually what sits underneath the file access mechanisms discussed in the previous chapter.

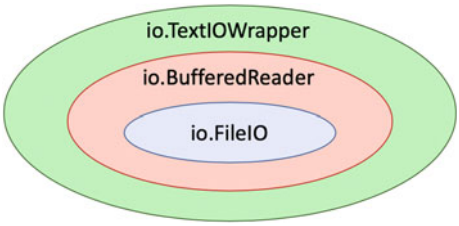
It is not actually necessary to understand this model to be able to read and write data to and from a file, however in some situations it is useful to have an understanding of this model so that you can modify the default behaviour when necessary.

The remainder of this chapter first introduces the Stream model, discusses Python streams in general and then presents the classes provided by Python. It then considers what is the actual effect of using the open() function presented in the last chapter.

19.2 What is a Stream?

Streams are objects which serve as sources or sinks of data. At first this concept can seem a bit strange. The easiest way to think of a stream is as a conduit of data flowing from or into a pool. Some streams read data straight from the "source of the data" and some streams read data from other streams. These latter streams then do some "useful" processing of the data such as converting the raw data into a specific format. The following figure illustrates this idea.

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-25943-3_19



In the above figure the initial `FileIO` stream reads raw data from the actual data source (in this case a file). The `BufferedReader` then buffers the data reading process for efficiency. Finally the `TextIOWrapper` handles string encoding; that is it converts strings from the typical ASCII representation used in a file into the internal representation used by Python (which uses Unicode).

You might ask at this point why have a streams model at all; after all we read and wrote data to files without needing to know about streams in the last chapter? The answer is that a stream can read or write data to or from a source of data rather than just from a file. Of course a file can be a source of data but so can a socket, a pipe, a string, a web service etc. It is therefore a more flexible data I/O model.

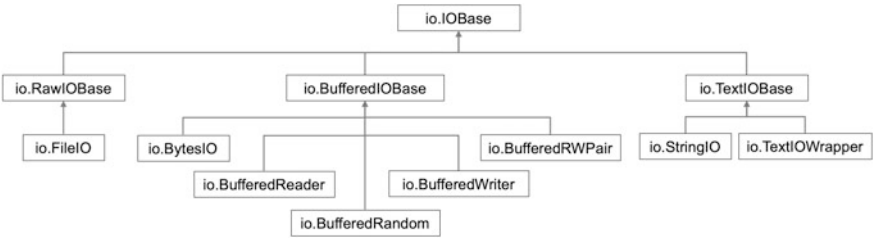
19.3 Python Streams

The Python `io` module provides Python's main facilities for dealing with data input and output. There are three main types of input/output these are text I/O, binary I/O and raw I/.O. These categories can be used with various types of data source/sinks.

Whatever the category, each concrete stream can have a number of properties such as being *read-only*, *write-only* or *read-write*. It can also support sequential access or random access depending on the nature of the underlying data sink. For example, reading data from a socket or pipe is inherently sequential where as reading data from a file can be performed sequentially or via a random access approach.

Whichever stream is used however, they are aware of the type of data they can process. For example, attempting to supply a string to a binary write-only stream will raise a `TypeError`. As indeed will presenting binary data to a text stream etc.

As suggested by this there are a number of different types of stream provided by the Python `io` module and some of these are presented below:



The abstract `IOBase` class is at the root of the stream IO class hierarchy. Below this class are stream classes for unbuffered and buffered IO and for text oriented IO.

19.4 IOBase

This is the abstract base class for all I/O stream classes. The class provides many abstract methods that subclasses will need to implement.

The `IOBase` class (and its subclasses) all support the iterator protocol. This means that an `IOBase` object (or an object of a subclass) can iterate over the input data from the underling stream.

`IOBase` also implements the Context Manager Protocol and therefore it can be used with the `with` and `with-as` statements.

The `IOBase` class defines a core set of methods and attributes including:

- `close()` flush and close the stream.
- `closed` an attribute indicating whether the stream is closed.
- `flush()` flush the write buffer of the stream if applicable.
- `readable()` returns True if the stream can be read from.
- `readline(size=-1)` return a line from the stream. If size is specified at most size bytes will be read.
- `readline(hint=-1)` read a list of lines. If hint is specified then it is used to control the number of lines read.
- `seek(offset[, whence])` This method moves the current the stream position/pointer to the given offset. The meaning of the offset depends on the whence parameter. The default value for whence is SEEK_SET.
- SEEK_SET or 0: seek from the start of the stream (the default); offset must either be a number returned by TextIOBase.tell(), or zero. Any other offset value produces undefined behaviour.
- SEEK_CUR or 1: “seek” to the current position; offset must be zero, which is a no-operation (all other values are unsupported).
- SEEK_END or 2: seek to the end of the stream; offset must be zero (all other values are unsupported).

- `seekable()` does the stream support seek().
- `tell()` return the current stream position/pointer.
- `writeable()` returns true if data can be written to the stream.
- `writelines(lines)` write a list of lines to the stream.

19.5 Raw IO/UnBuffered IO Classes

Raw IO or unbuffered IO is provided by the `RawIOBase` and `FileIO` classes.

RawIOBase This class is a subclass of `IOBase` and is the base class for raw binary (aka unbuffered) I/O. Raw binary I/O typically provides low-level access to an underlying OS device or API, and does not try to encapsulate it in high-level primitives (this is the responsibility of the Buffered I/O and Text I/O classes that can wrap a raw I/O stream). The class adds methods such as:

- `read(size=-1)` This method reads up to size bytes from the stream and returns them. If size is unspecified or -1 then all available bytes are read.
- `readall()` This method reads and returns all available bytes within the stream.
- `readint(b)` This method reads the bytes in the stream into a pre-allocated, writable bytes-like object b (e.g. into a byte array). It returns the number of bytes read.
- `write(b)` This method writes the data provided by `b` (a bytes -like object such as a byte array) into the underlying raw stream.

FileIO The `FileIO` class represents a raw unbuffered binary IO stream linked to an operating system level file. When the `FileIO` class is instantiated it can be given a file name and the mode (such as 'r' or 'w' etc.). It can also be given a flag to indicate whether the file descriptor associated with the underlying OS level file should be closed or not.

This class is used for the low-level reading of binary data and is at the heart of all file oriented data access (although it is often wrapped by another stream such as a buffered reader or writer).

19.6 Binary IO/Buffered IO Classes

Binary IO aka Buffered IO is a filter stream that wraps a lower level `RawIOBase` stream (such as a `FileIO` stream). The classes implementing buffered IO all extend the `BufferedIOBase` class and are:

BufferedReader When reading data from this object, a larger amount of data may be requested from the underlying raw stream, and kept in an internal buffer. The buffered data can then be returned directly on subsequent reads.

BufferedWriter When writing to this object, data is normally placed into an internal buffer. The buffer will be written out to the underlying `RawIOBase` object under various conditions, including:

- when the buffer gets too small for all pending data;
- when `flush()` is called;
- when the `BufferedWriter` object is closed or destroyed.

BufferedRandom A buffered interface to random access streams. It supports `seek()` and `tell()` functionality.

BufferedRWPair A buffered I/O object combining two unidirectional RawIOBase objects – one readable, the other writeable—into a single bidirectional endpoint.

Each of the above classes wrap a lower level byte oriented stream class such as the `io.FileIO` class, for example:

```
f = io.FileIO('data.dat')
br = io.BufferedReader(f)
print(br.read())
```

This allows data in the form of bytes to be read from the file 'data.dat'. You can of course also read data from a different source, such as an in memory `BytesIO` object:

```
binary_stream_from_file =
io.BufferedReader(io.BytesIO(b'starship.png'))
bytes = binary_stream_from_file.read(4)
print(bytes)
```

In this example the data is read from the `BytesIO` object by the `BufferedReader`. The `read()` method is then used to read the first 4 bytes, the output is:

```
b'star'
```

Note the 'b' in front of both the string 'starship.png' and the result 'star'. This indicates that the string literal should become a bytes literal in Python 3. Bytes literals are always prefixed with 'b' or 'B'; they produce an instance of the bytes type instead of the str type. They may only contain ASCII characters.

The operations supported by buffered streams include, for reading:

- `peek(n)` return up to *n* bytes of data without advancing the stream pointer. The number of bytes returned may be less or more than requested depending on the amount of data available.
- `read(n)` return n bytes of data as bytes, if n is not supplied (or is negative) the read all available data.
- `readl(n)` read up to n bytes of data using a single call on the raw data stream.

The operations supported by buffered writers include:

- `write(bytes)` writes the bytes-like data and returns the number of bytes written.
- `flush()` This method forces the bytes held in the buffer into the raw stream.

19.7 Text Stream Classes

The text stream classes are the `TextIOBase` class and its two subclasses `TextIOWrapper` and `StringIO`.

TextIOBase This is the root class for all Text Stream classes. It provides a character and line based interface to Stream I/O. This class provides several additional methods to that defined in its parent class:

- `read(size=-1)` This method will return at most size characters from the stream as a single string. If size is negative or None, it will read all remaining data.
- `readline(size=-1)` This method will return a string representing the current line (up to a newline or the end of the data whichever comes first). If the stream is already at EOF, an empty string is returned. If size is specified, at most size characters will be read.
- `seek(offset, [, whence])` change the stream position/pointer by the specified offset. The optional `whence` parameter indicates where the seek should start from:
  - `SEEK_SET` or 0: (the default) seek from the start of the stream.
  - `SEEK_CUR` or 1: seek to the current position; offset must be zero, which is a no-operation.
  - `SEEK_END` or 2: seek to the end of the stream; offset must be zero.
- `tell()` Returns the current stream position/pointer as an opaque number. The number does not usually represent a number of bytes in the underlying binary storage.
- `write(s)` This method will write the string s to the stream and return the number of characters written.

TextIOWrapper. This is a buffered text stream that wraps a buffered binary stream and is a direct subclass of `TextIOBase`. When a `TextIOWrapper` is created there are a range of options available to control its behaviour:

```
io.TextIOWrapper(buffer, encoding=None, errors=None, newline=No
ne, line_buffering=False, write_through=False)
```

Where

- `buffer` is the buffered binary stream.
- `encoding` represents the text encoding used such as UTF-8.
- `errors` defines the error handling policy such as *strict* or *ignore*.
- `newline` controls how line endings are handled for example should they be ignored (None) or represented as a linefeed, carriage return or a newline/carriage return etc.
- `line_buffering` if `True` then `flush()` is implied when a call to write contains a newline character or a carriage return.
- `write_through` if `True` then a call to write is guaranteed not to be buffered.

The `TextIOWrapper` is wrapped around a lower level binary buffered I/O stream, for example:

```
f = io.FileIO('data.txt')
br = io.BufferedReader(f)
text_stream = io.TextIOWrapper(br, 'utf-8')
```

StringIO This is an in memory stream for text I/O. The initial value of the buffer held by the `StringIO` object can be provided when the instance is created, for example:

```
in_memory_text_stream = io.StringIO('to be or not to be that is
the question')
print('in_memory_text_stream', in_memory_text_stream)
print(in_memory_text_stream.getvalue())
in_memory_text_stream.close()
```

This generates:

```
in_memory_text_stream <_io.StringIO object at 0x10fdfaee8>
to be or not to be that is the question
```

Note that the underlying buffer (represented by the string passed into the `StringIO` instance) is discarded when the `close()` method is called.

The `getvalue()` method returns a string containing the entire contents of the buffer. If it is called after the stream was closed then an error is generated.

19.8 Stream Properties

It is possible to query a stream to determine what types of operations it supports. This can be done using the `readable()`, `seekable()` and `writeable()` methods. For example:

```
f = io.FileIO('myfile.txt')
br = io.BufferedReader(f)
text_stream = io.TextIOWrapper(br, encoding='utf-8')

print('text_stream', text_stream)
print('text_stream.readable():', text_stream.readable())
print('text_stream.seekable()', text_stream.seekable())
print('text_stream.writeable()', text_stream.writable())

text_stream.close()
```

The output from this code snippet is:

```
text_stream <_io.TextIOWrapper name='myfile.txt' encoding='utf-
8'>
text_stream.readable(): True
text_stream.seekable() True
text_stream.writeable() False
```

19.9 Closing Streams

All opened streams must be closed. However, you can close the top level stream and this will automatically close lower level streams, for example:

```
f = io.FileIO('data.txt')
br = io.BufferedReader(f)
text_stream = io.TextIOWrapper(br, 'utf-8')
print(text_stream.read())
text_stream.close()
```

19.10 Returning to the open() Function

If streams are so good then why don't you use them all the time? Well actually in Python 3 you do! The core open function (and indeed the `io.open()` function) both return a stream object. The actual type of object returned depends on the file mode specified, whether buffering is being used etc. For example:

```
import io

# Text stream
f1 = open('myfile.txt', mode='r', encoding='utf-8')
print(f1)

# Binary IO aka Buffered IO
f2 = open('myfile.dat', mode='rb')
print(f2)

f3 = open('myfile.dat', mode='wb')
print(f3)

# Raw IO aka Unbufferedf IO
f4 = open('starship.png', mode='rb', buffering=0)
print(f4)
```

When this short example is run the output is:

```
<_io.TextIOWrapper name='myfile.txt' mode='r' encoding='utf-8'>
<_io.BufferedReader name='myfile.dat'>
<_io.BufferedWriter name='myfile.dat'>
<_io.FileIO name='starship.png' mode='rb' closefd=True>
```

As you can see from the output, four different types of object have been returned from the `open()` function. The first is a `TextIOWrapper`, the second a `BufferedReader`, the third a BufferedWriter and the final one is a `FileIO` object. This reflects the differences in the parameters passed into the open (0 function. For example, f1 references a `io.TextIOWrapper` because it must encode (convert) the input text into Unicode using the UTF-8 encoding scheme. While f2 holds a io.`BufferedReader` because the mode indicates that we want to read binary data while f3 holds a io.`BufferedWriter` because the mode used indicates we want to write binary data. The final call to open returns a `FileIO` because we have indicated that we do not want to buffer the data and thus we can use the lowest level of stream object.

In general the following rules are applied to determine the type of object returned based on the modes and encoding specified:

Class	mode	Buffering
FileIO	binary	no
BufferedReader	‘rb’	yes
BufferedWriter	‘wb’	yes
BufferedRandom	‘rb+’ ‘wb+’ ‘ab+’	yes
TextIOWrapper	Any text	yes

Note that not all mode combinations make sense and thus some combinations will generate an error.

In general you don't therefore need to worry about which stream you are using or what that stream does; not least because all the streams extend the `IOBase` class and thus have a common set of methods and attributes.

However, it is useful to understand the implications of what you are doing so that you can make better informed choices. For example, binary streams (that do less processing) are faster than Unicode oriented streams that must convert from ASCII into Unicode.

Also understanding the role of streams in Input and Output can also allow you to change the source and destination of data without needing to re-write the whole of your application. You can thus use a file or stdin for testing and a socket for reading data in production.

19.11 Online Resources

See the following online resources for information on the topics in this chapter:

- https://docs.python.org/3/library/io.html. This provides the Python Standard Library guide to the core tools available for working with streams.

19.12 Exercise

Use the underlying streams model to create an application that will write binary data to a file. You can use the 'b' prefix to create a binary literal to be written, for example b 'Hello World'.

Next create another application to reload the binary data from the file and print it out.